IBM FORTRAN/2™ Computer Language Series

Compile, Link and Run

Programming Family

IBM

84X1754

IBM FORTRAN/2™ Computer Language Series

Compile, Link and Run

Programming Family

IBM

First Edition (September, 1987)

Requests for copies of this publication and for technical information about IBM Personal Computer products should be made to your authorized IBM Personal Computer dealer or your IBM Marketing Representative.

Preface

Library Guide

The IBM FORTRAN/2 library consists of three manuals. The following chart shows where different information is located.

If You Want to...	Refer to...
Install the product	Compile, Link, and Run
Learn basic facts about the language	Fundamentals
Know the syntax of an instruction	Language Reference
Understand error messages	Language Reference
Debug a program	Compile, Link, and Run
Compile a program	Compile, Link, and Run
Link a program	Compile, Link, and Run
Write a program	Fundamentals, Language Reference, and Compile, Link, and Run

Inside This Book

This manual explains how to compile, link, and run programs written in IBM FORTRAN/2, a FORTRAN language that can be used with the IBM Math Co-Processor. By increasing the speed of all numeric and trigonometric functions, the math coprocessor makes IBM FORTRAN/2 ideal for scientific and technical use. Although written to take advantage of the math coprocessor, IBM FORTRAN/2 provides a compiler option that emulates the functions of the coprocessor.

IBM FORTRAN/2 is designed according to the specifications of the following industry standards as understood and interpreted by IBM as of September, 1987:

- American National Programming Language FORTRAN 77 (ANSI X3.9-1978)
- The coded character sets—7-bit American National Standard for Information Exchange (ANSI X3.4-1986)
- The Code Extension Techniques for use with the 7-bit Coded Character Set of American National Standard for Information Exchange (ANSI 3.4-1974)
- The IEEE Standard 754 for floating point arithmetic, with the following differences:
  - Rounds to nearest mode only
  - No rounding precision mode
  - No trapping (signaling) NaNs (not a number)
  - No user traps
  - Exception status flags are not supported

In addition, IBM FORTRAN/2 contains many useful extensions to that language beyond the ANSI X3.9-1978 standard. This manual explains those extensions as well.

Audience Statement

This manual is designed for people who have programmed previously and who are familiar with IBM Personal Computer Disk Operating System (DOS) or IBM Operating System/2™ (OS/2™).[1] If you are not familiar with either of these operating systems, refer to the IBM *DOS Reference* manual or *OS/2 User's Reference* manual for information about them.

[1] Operating System/2 and OS/2 are trademarks of the International Business Machines Corporation.

Contents

Chapter 1. Introduction

About This Book

This manual is a part of the IBM FORTRAN/2™[1] documentation set. Its purpose is to assist you in the installation and use of the IBM FORTRAN/2 language. The manual is organized into the following chapters:

- Chapter 1, "Introduction" describes the purpose of the manual and identifies the files and resources you need to successfully write, compile, link, run, and debug programs on your IBM personal computer using the IBM FORTRAN/2 compiler.
- Chapter 2, "Installation" describes how to install the IBM FORTRAN/2 compiler, create work files, and edit the configuration file. The chapter also explains the procedures for compiling, linking, and running a sample IBM FORTRAN/2 program.
- Chapter 3, "Compiling Your Program" describes in more detail how to compile the contents of a source file to create an object module, obtain a compiler listing (including diagnostics), and redirect a compiler listing to a desired disk file.
- Chapter 4, "LINK: The Object Linker" describes the actions performed by the linker and identifies the IBM FORTRAN/2 Libraries that can be used with an object module.
- Chapter 5, "Running Your Program" describes how to start and cancel execution of a program under DOS and OS/2, use IBM FORTRAN/2 runtime routines, react to runtime errors, and control various I/O devices during runtime.
- Chapter 6, "LIB: The Library Manager" describes how to use the library manager to store and retrieve object modules.
- Chapter 7, "Debugging Your Program" describes how to set up your configuration to accept Debug commands and display Debug output, stop a program and enter a Debug command, and efficiently use Debug commands and Debug device defaults.

[1] FORTRAN/2 is a trademark of the International Business Machines Corporation.

- Chapter 8, "Interfaces with Other IBM Languages and Products" describes how to define subprograms written in assembler and how to use OS procedures.

Note: This manual is not a tutorial; rather, each section explains a particular aspect of IBM FORTRAN/2 or its use.

Additional Documentation

The other manuals of this documentation set are the *IBM FORTRAN/2 Fundamentals* manual and the *IBM FORTRAN/2 Language Reference* manual. These manuals are organized as follows:

IBM FORTRAN/2 Fundamentals:

- Chapter 1, "Introduction," describes the purpose of the manual and identifies the files and resources you need to successfully write, compile, link, run, and debug programs using the IBM FORTRAN/2 compiler.
- Chapter 2, "General Information," describes IBM FORTRAN/2 data types, names, expressions, and coding conventions.
- Chapter 3, "Program Structure," describes the relationship between the main program unit and all forms of subprograms.
- Chapter 4, "File Processing," describes the IBM FORTRAN/2 file system and provides information about input and output.

IBM FORTRAN/2 Language Reference:

- Chapter 1, "Introduction," describes the purpose of the manual and identifies the files and resources you need to successfully write, compile, link, run, and debug programs using the IBM FORTRAN/2 compiler.
- Chapter 2, "Statements," provides a complete description of the statements that are available for use with the IBM FORTRAN/2 language — including the purpose and correct coding for each statement.

- Chapter 3, "I/O Editing," is a guide for the use of the editing commands available in the IBM FORTRAN/2 language.
- Chapter 4, "Portability, Conversion, and Extensions" describes the issues related to using the IBM FORTRAN/2 compiler to compile existing FORTRAN programs and to run programs compiled with the IBM FORTRAN/2 compiler on different computers.
- Appendix A, "Messages" lists the messages and codes for errors and warnings that are reported by the compiler, library manager, linker, runtime system, and debug facilities.
- Appendix B. "Floating Point Operations and Exceptional Values", contains tables and references that describe floating-point operations and explain how exceptional values (infinities and NaNs) are generated and written.
- Appendix C, " Code Optimization", describes the code optimization performed by the IBM FORTRAN/2 compiler.
- Appendix D, "Intrinsic Functions" contains a complete table of IBM FORTRAN/2 intrinsic functions and notes on using them.
- Appendix E, "Additional Routines", contains information about routines provided that may be helpful when using IBM FORTRAN/2.
- Appendix F, "Internal Data Representation", describes the internal representation of data.
- Appendix G. "Limits and Ranges" lists the limits and ranges of IBM FORTRAN/2 programs, I/O, and data.
- Appendix H, "ASCII Codes" lists all the ASCII codes (in decimal) and their associated characters.
- Appendix I, "Hollerith and Hexadecimal Data" contains instructions for using Hollerith and hexadecimal data types.

Note: A glossary of terms used with IBM FORTRAN/2 follows the appendixes.

What You Need

To successfully write, compile, link, run, and debug programs on your IBM personal computer using the IBM FORTRAN/2 compiler, you need:

- IBM FORTRAN/2 master diskettes.
- IBM Personal Computer Disk Operating System (DOS), Version 3.30; or IBM Operating System/2 (OS/2), Version 1.0.
- One of the following systems:
  - IBM Personal Computer
  - IBM Personal Computer XT™[2]
  - IBM Personal Computer AT®[3]
  - IBM Personal Computer PC Convertible
  - IBM Personal System/2™ Models 30, 50, 60, or 80.

  With DOS, a minimum of 320KB available user memory is required. With OS/2, a minimum of 1.5MB available user memory is needed.
- Monitor with 80-column capacity.
- A math coprocessor compatible with your system (optional, but recommended).
- A fixed disk and one 1.2MB, 1.44MB, 720KB, or 360KB diskette drive; *or* two diskette drives, each of which can be 1.2MB, 1.44MB, 720KB, or 360KB. A fixed disk is recommended.
- A printer (optional, but recommended).

---

[2] Personal Computer XT and Personal System/2 are trademarks of the International Business Machines Corporation.

[3] Personal Computer AT is a registered trademark of International Business Machines Corporation.

What You Have

IBM FORTRAN/2 Software

IBM FORTRAN/2 software is provided on four 360KB (5¼") diskettes and two 720KB (3½") diskettes. These are referred to as the master diskettes. The 360KB diskettes are:

IBM FORTRAN/2 "INSTALL" Master Diskette (360KB)

Filename	Contents
README	A list of changes to IBM FORTRAN/2 not included in the manuals
PACKING.LST	Packing list of delivered software
INSTALL.EXE	Installation program for DOS
FORTRAN.EXE	IBM FORTRAN/2 Compiler
FORTRAN.CER	Compiler Error Messages
DEMO.FOR	Demonstration Program
DEMO.DCM	Demonstration Program Debug Command Input File
QFORT.BAT	Sample Compiler Batch File
QFORTLNK.BAT	Sample Link Batch File
QFORTRUN.BAT	Sample Runtime Batch File

IBM FORTRAN/2 "LINK_RUN" Master Diskette (360KB)

Filename	Contents
FORTRAN.ERR	Runtime Messages
FORTRAN.DER	Debug Messages
FORTDBG.HLP	Debug online help facility
FORTRUN.DLL	Dynamic runtime and debug
FORTRUE.DLL	Dynamic runtime, debug, and emulator
LINK.EXE	Linker
LINK.PIF	Linker Program Information File for TopView
LIB.PIF	Library Manager Program Information File for TopView
FORTILC.ASM	Assembly source for inter-language calls
COPYMEM.ASM	Assembly source for copying memory
FORTRAN.PIF	Compiler Program Information File for TopView

IBM FORTRAN/2 "NP_LIBRARY" Master Diskette (360KB)

Filename	Contents
FORTRRN.LIB	Runtime and Debug library
FORTRIN.LIB	Intrinsic function library
FORTRDN.LIB	Runtime and Debug import library
PCDOS.LIB	OS/2 to DOS mappings library
LIB.EXE	Library Manager
FORTRAN.PIP	Program Information Profile for OS/2
FORTPIP.LIB	Program Information Profile library

IBM FORTRAN/2 "EM_LIBRARY" Master Diskette (360KB)

Filename	Contents
FORTRRE.LIB	Runtime, Debug, and emulator library
FORTRIE.LIB	Intrinsic function library
FORTRDE.LIB	Runtime, Debug, and emulator import library
PCDOS.LIB	OS/2 to DOS mappings library
LIB.EXE	Library Manager

The 720KB (3½") diskettes provided with IBM FORTRAN/2 are:

IBM FORTRAN/2 "INSTALL" Master Diskette (720KB)

Includes the contents of the 360KB IBM FORTRAN/2 "INSTALL" and IBM FORTRAN/2 "LINK_RUN" master diskettes.

IBM FORTRAN/2 "LIBRARY" Master Diskette (720KB)

Includes the contents of the 360KB IBM FORTRAN/2 "NP_LIBRARY" and IBM FORTRAN/2 "EM_LIBRARY" master diskettes.

Summary of Changes

Listed below are features of IBM FORTRAN/2 that differ from the predecessor product, IBM Professional FORTRAN.

- A single copy of the compiler that runs in all modes of OS/2 and DOS Version 3.30.
- Generated code that can be linked for all modes of OS/2 and DOS Version 3.30.
- Code segments of compiler and generated code that can be shared.
- Calls that can be made to the OS/2 Application Programming Interface (OS/2 only).
- Interactive Debugger with enhanced display capabilities and new commands.
- New compiler options and environment variables.
- Mainframe compatibility enhancements.
- Isolation of all textual messages to facilitate language translation.
- Support for file sharing over the IBM PC Network and in a multitasking environment on OS/2.
- IBM Math Co-Processor emulation.
- Installation aids for various system configurations.
- Overlay support for DOS.
- IMPLICIT NONE statement.
- EJECT statement.
- Increased limits and ranges.
- A compiler option to create code specific to various system configurations.
- Threaded compiler error messages.
- Backslash edit control descriptor.
- Minus carriage control character.

- Comma external field terminator.
- OPEN with ACCESS='APPEND'.
- Alternate version of the SNGL function that forces rounding to single precision.
- New DSNGL function that forces rounding to double precision.
- Improved Assembly Language support (FORTILC).
- Alternate character length function (CLEN).
- Additional compiler listing options.
- Compiler status messages.
- COMPLEX*16.
- Mixing of complex and double precision values in the same expression.
- Additional code optimization.
- Underscore allowed as a non-leading character in a symbolic name.
- $ as high end of implicit range.

Chapter 2. Installation

If you want to install IBM FORTRAN/2 and compile, link, and run a sample program, read this chapter. It explains how to:

- Install IBM FORTRAN/2, following instructions appropriate for your system.
- Compile the sample program using appropriate compiler options.
- Link the sample program.
- Run the sample program once to display output on your screen, and a second time to redirect output to your printer.

If IBM FORTRAN/2 is already installed and you do not want to process the sample application, skip this chapter and go on to Chapter 3, "Compiling Your Program." It explains how to use the compiler in more detail.

Your Basic System Configuration

The configuration of your computer system is the assembled set of hardware and software that makes up your diskette or fixed-disk system. Before you can develop or run IBM FORTRAN/2 programs, your configuration must meet or exceed the requirements described in Chapter 1, "Introduction." Refer to "What You Need" on page 1-4 to be sure that you have all of the hardware and software that IBM FORTRAN/2 requires.

Installation Programs

IBM FORTRAN/2 can be installed under OS/2 or DOS. Separate installation programs are provided on the IBM FORTRAN "INSTALL" master diskette.

There are 5¼"and 3½" versions of this diskette. Use the 5¼" diskette if you have a 1.2MB or 360KB diskette drive. Use the 3½" diskette if you have a 1.44MB or 720KB diskette drive.

Under OS/2, IBM FORTRAN/2 is installed using the installation aid program INSTAID. How to start INSTAID is explained in "Installing Using OS/2" on page 2-5. Under DOS, installation can be performed using INSTALL.EXE. How to start this program is explained in "Installing and Using DOS" on page 2-5.

The INSTAID and INSTALL.EXE programs are interactive. This means that they display prompts to guide you through the installation process. To get an idea of the kinds of prompts that are displayed and the types of responses that are necessary, read "Installation Prompts" on page 2-6 *before* starting INSTAID or INSTALL.EXE.

Under DOS, IBM FORTRAN/2 can also be installed using the Network Installation Aid. Considerations for its use are explained in "Installing IBM FORTRAN/2 for Use on PC Local Area Network" on page 2-16.

When IBM FORTRAN/2 is installed under DOS, it can be added to TopView. The procedure for doing so is explained in "Adding IBM FORTRAN/2 to TopView" on page 2-16.

Note: If you installing to a fixed disk, the method you select for installation is not important as long as it is compatible with your OS/2 or DOS system. You should choose the method with which you are most comfortable.

If you are installing to diskettes, install IBM FORTRAN/2 with INSTALL.EXE. After it is installed, IBM FORTRAN/2 will work in *any* of the environments indicated above.

Backing Up the Master Diskettes

The master diskettes containing IBM FORTRAN/2 software should be copied for backup purposes. To save time, this should be done *before* you begin installation.

The master diskettes can be copied with the DISKCOPY command. To prevent accidental erasure of the master diskettes, be sure that they are write-protected before starting the copy operation. The DISKCOPY command is explained in the *OS/2 User's Reference* manual and the *DOS Reference* manual.

If you do not copy the diskettes beforehand, the INSTAID and INSTALL.EXE programs will allow you to do so during installation. This option is selected by responding to a prompt.

Latest Information

A README file is provided on the IBM FORTRAN/2 "INSTALL" master diskette. It describes the latest changes to IBM FORTRAN/2. You should read the file to familiarize yourself with those changes.

You can read the file during installation. The INSTAID or INSTALL.EXE program will prompt you for this purpose. To save time, however, you should read the file *before* starting installation. To do so, enter the following command:

```
MORE <n:README
```

where *n* is the letter of the drive containing the diskette.

You should also display the PACKING.LST file on the IBM FORTRAN/2 "INSTALL" master diskette to verify that you have received all required diskettes and files. To do this enter:

```
MORE <n:PACKING.LST
```

Again, *n* is the letter of the drive containing the diskette.

When you read the README and PACKING.LST files, you should also direct them to the printer so that they can be easily referenced at a later time.

See the *OS/2 User's Reference* manual or *DOS Reference* manual for information about the MORE command and instructions for directing files to the printer.

Installation Requirements

Time Requirements

Installation of IBM FORTRAN/2 will take about 10 minutes if you install only one library. Installation may take up to 30 minutes if you install all libraries.

Disk Space Requirements

Installation on a fixed disk with all the libraries requires approximately 1.7MB of disk space. Installation on a fixed disk with only the library for DOS and the IBM Math Co-Processor requires approximately 800KB of disk space.

Diskette Requirements

If you are not installing on a fixed disk, you will need up to four 1.2MB (5¼") diskettes, four 1.44MB diskettes (3½"), four 720KB (3½") diskettes, or seven 360KB (5¼") diskettes, depending on the type of diskette drive you have and the libraries that are installed. Although the INSTAID or INSTALL.EXE installation program will ask if you want it to format the diskettes, you should format them before beginning installation to reduce the time required to install IBM FORTRAN/2.

You can use the FORMAT command to format the diskettes. For information about the FORMAT command see the *OS/2 User's Reference* manual or the *DOS Reference* manual.

Starting Installation

Installing Using OS/2

To begin installation of IBM FORTRAN/2 under OS/2 mode, restart the system and make sure your path includes

```
C:\;C:\OS2\INSTALL
```

Insert your copy of the IBM FORTRAN/2 "NP_LIBRARY" 5-¼" master diskette or the "LIBRARY" 3-1/2" master diskette into drive A. Enter the following command at the C: prompt:

```
INSTAID
```

The INSTAID program will execute. Select the option to install a program.

After the installation is complete, change CONFIG.SYS and STARTUP.CMD as indicated in "After Installation" on page 2-17, then restart the system. At the Program Selector Screen, select `OS/2 Command Prompt` , then change the directory to the path where you installed IBM FORTRAN/2. You can then execute IBM FORTRAN/2.

See "Compiling the Sample Program" on page 2-22 for more information.

Installing Using DOS

To begin the installation process under DOS, insert your copy of the IBM FORTRAN "INSTALL" diskette into one of the drives. Enter the following command at the C: prompt:

```
n:INSTALL
```

where *n* is the letter of the drive containing the diskette.

The INSTALL.EXE program will execute.

Installation Prompts

When the INSTAID or INSTALL.EXE program is executed, it will display prompts. The prompts will guide you through the installation process, during which you will:

- Create backup copies of the IBM FORTRAN master diskettes (if you did not do so earlier)
- Format working diskettes (if you did not do so earlier)
- Create working copies of various files and libraries from the IBM FORTRAN master diskettes.

You will have to respond to the prompts by giving information about the type of system you are using, pathname assignments, and your choice of libraries. Read the prompts carefully and respond as appropriate.

README File

The installation program will ask if you wish to see and print the README file to obtain the latest information about IBM FORTRAN/2. To save time, you should read and print its contents before invoking the installation program. For information on how to read and print the README file, see "Latest Information" on page 2-3.

Diskette Backup

The installation program will give you the opportunity to backup the IBM FORTRAN/2 master diskettes during installation. If you select this option, the program will use the DISKCOPY command to produce copies of the diskettes. Again, to save time, you should backup the diskettes before invoking the INSTAID or INSTALL.EXE installation program. For information on how to backup the IBM FORTRAN/2 master diskettes, see "Backing Up the Master Diskettes" on page 2-2.

Fixed Disk or Diskette

To determine the type of system you have, the installation program will ask you to specify if it has dual diskettes or a fixed-disk. Indicate your particular configuration.

Drive Letter

The program will ask you to specify the destination drive. Respond by giving its letter. For example, B: for a diskette installation, and C: for a fixed disk installation.

Network (Fixed Disk Only)

The program will ask if you want to install for a network.

Choice of Directory (Fixed Disk Only)

The installation program will ask you to specify the directory in which you want IBM FORTRAN/2 installed. You can enter a partially or fully qualified pathname of a directory. The default for a non-network installation is

```
\FORTRAN
```

The default for a network installation is:

```
\APPS\FORTRAN
```

The pathname should *not* include a drive letter. The drive letter is given in response to a previous prompt.

Note: For installation on diskettes, the files are installed in the root directories.

Choice of Libraries

IBM FORTRAN/2 provides a set of libraries for OS/2, DOS, and Family Application Programming Interface. There are two libraries in each set: one for use with the IBM Math Co-Processor (FORTRAN.LIB), and one for use with the emulator (FORTRAE.LIB). The installation program will ask you to identify the libraries you want.

Although you can install all six libraries, in most cases you will need only one. Identify the library (or libraries) to be installed in response to the prompts. If you select only the emulation library, you must use the /N option for compilation. For additional information on selecting libraries, see Chapter 4, "LINK: The Object Linker."

Format Diskettes (Diskettes Only)

The installation program will give you the opportunity to format working diskettes during installation. If you select this option, the program will use the FORMAT command for this purpose.

As recommended earlier, you should format the diskettes before starting installation to save time during installation. See "Diskette Requirements" on page 2-4 for information about how to format the work diskettes.

Work Diskettes

The work diskettes are needed during installation to store copies of various files and libraries from the IBM FORTRAN/2 master diskettes. The number of work diskettes that are created depends on the type of diskette drive used with the system and the libraries requested. You will use the work diskettes to create, compile, link, run, and debug your programs. You should not use the master diskettes for these purposes.

720KB, 1.2MB or 1.44MB Diskettes

The following work diskettes are created for installation using 720KB, 1.2MB, or 1.44MB diskettes. If you are using 1.44MB diskettes, the layout is the same as for 720KB diskettes.

IBM FORTRAN/2 "COMPILER" Work Diskette

This diskette is used to compile programs, run programs, and create libraries.

Filename	Contents
FORTRAN.EXE	Compiler
FORTRAN.CER	Compiler messages
FORTRAN.ERR	Runtime messages
FORTRAN.DER	Debug messages

FORTDBG.HLP	Debug help file
LIB.EXE	Library Manager
DEMO.FOR	Demonstration program
DEMO.DCM	Demonstration debug command input file
QFORT.BAT	Sample compiler batch file
QFORTLNK.BAT	Sample link batch file
QFORTRUN.BAT	Sample run batch file
FORTRUN.DLL	Dynamic link runtime and debug for IBM Math Co-Processor (only if installed for OS/2 mode and math coprocessor)
FORTRUE.DLL	Dynamic link runtime, debug and emulator for math coprocessor emulation (only if installed for OS/2 mode and emulation)

IBM FORTRAN/2 "PLIB" Work Diskette

This diskette is used when linking for OS/2 mode or Family Application Programming Interface. It is created only if OS/2 mode libraries are requested.

Filename	Contents
FORTRAN.LIB	Objects for intrinsic functions, and imports for run-time and debug (only installed if math coprocessor libraries are requested)
FORTRAE.LIB	Objects for intrinsic functions and imports for run-time, debug, and emulator (only installed if emulation libraries are requested)
LINK.EXE	Linker

IBM FORTRAN/2 "RLIB" Work Diskette

This diskette is used when linking for DOS or for OS/2 real mode. It is created only if DOS mode libraries are requested.

Filename	Contents
FORTRAN.LIB	Objects for intrinsic functions, runtime, debug, and OS/2 to DOS mappings (only installed if math coprocessor libraries are requested)
FORTRAE.LIB	Objects for intrinsic functions, runtime, debug, emulator, and OS/2 to DOS mappings (only installed if emulation libraries are requested)
LINK.EXE	Linker

IBM FORTRAN/2 "FLIB" Work Diskette

This diskette is used when linking or binding Family Application Programming Interface applications. It is created only if Family Application Programming Interface libraries are requested.

Filename	Contents
FORTRAN.LIB	Objects for intrinsic functions, runtime, and debug (only installed if math coprocessor libraries are requested)
FORTRAE.LIB	Objects for intrinsic functions, runtime, debug, and emulator (only installed if emulation libraries are requested)
LINK.EXE	Linker

360KB Work Diskettes

The following work diskettes are created for installation using 360KB diskettes.

IBM FORTRAN/2 "COMPILER" Work Diskette (360KB)

This diskette is used to compile programs and create libraries.

Filename	Contents
FORTRAN.EXE	Compiler

FORTRAN.CER	Compiler messages
LIB.EXE	Library manager
DEMO.FOR	Demonstration program
DEMO.DCM	Demonstration debug command input file
QFORT.BAT	Sample compiler batch file
QFORTLNK.BAT	Sample link batch file
QFORTRUN.BAT	Sample run batch file

IBM FORTRAN/2 "RUNTIME" Work Diskette (360KB)

This diskette is used to run programs.

Filename	Contents
FORTRAN.ERR	Runtime messages
FORTRAN.DER	Debug messages
FORTDBG.HLP	Debug help file
FORTRUN.DLL	Dynamic link runtime and debug for IBM Math Co-Processor (only if installed for OS/2 mode and math coprocessor)
FORTRUE.DLL	Dynamic link runtime, debug and emulator for math coprocessor emulation (only if installed for OS/2 mode and emulation)

IBM FORTRAN/2 "PLIB" Work Diskette (360KB)

This diskette is used when linking OS/2 mode or Family Application Programming Interface applications. It is created only if OS/2 mode libraries are requested.

Filename	Contents
FORTRAN.LIB	Objects for intrinsic functions, and imports for run-time and debug (only installed if math coprocessor libraries are requested)
FORTRAE.LIB	Objects for intrinsic functions and imports for run-time, debug, and emulator (only installed if emulation libraries are requested)
LINK.EXE	Linker

IBM FORTRAN/2 "RLIBN" Work Diskette (360KB)

This diskette is used when linking for DOS mode for use with the IBM Math Co-Processor. It is created only if real mode or DOS libraries and math coprocessor libraries are requested.

Filename	Contents
FORTRAN.LIB	Objects for intrinsic functions, runtime, debug, and OS/2 to DOS mappings.
LINK.EXE	Linker

IBM FORTRAN/2 "RLIBE" Work Diskette (360KB)

This diskette is used when linking for DOS mode for use with the emulator. It is created only if real mode or DOS libraries and emulation libraries are requested.

Filename	Contents
FORTRAE.LIB	Objects for intrinsic functions, runtime, debug, emulator, and OS/2 to DOS mappings.
LINK.EXE	Linker

IBM FORTRAN/2 "FLIBN" Work Diskette (360KB)

This diskette is used when linking or binding Family Application Programming Interface applications for use with the math coprocessor. It is created only if Family Application Programming Interface libraries and math coprocessor libraries are requested.

Filename	Contents
FORTRAN.LIB	Objects for intrinsic functions, runtime, and debug
LINK.EXE	Linker

IBM FORTRAN/2 "FLIBE" Work Diskette (360KB)

This diskette is used when linking or binding Family Application Programming Interface applications for use with the emulator. It is created only if Family Application Programming Interface libraries and emulation libraries are requested.

Filename	Contents
FORTRAE.LIB	Objects for intrinsic functions, runtime, debug, and emulator
LINK.EXE	Linker

Work Directories (Fixed Disk Only)

The following work directories are created for installing IBM FORTRAN/2 onto a fixed disk. The default for a non-network installation is

```
\FORTRAN
```

The default for a network installation is

```
\APPS\FORTRAN
```

These directories contain the following:

File/Directory	Contents
PLIB	Directory for libraries to link for OS/2 mode (if requested) and Family Application Programming Interface applications.
RLIB	Directory for libraries to link for DOS mode (if requested)
FLIB	Directory for libraries to link or bind Family Application Programming Interface applications (if requested)
	The files in this directory are:
FORTRAN.EXE	Compiler
FORTRAN.CER	Compiler messages
FORTRAN.ERR	Runtime messages

FORTRAN.DER	Debug messages
FORTDBG.HLP	Debug help file
LIB.EXE	Library Manager
LINK.EXE	Linker
DEMO.FOR	Demonstration program
DEMO.DCM	Demonstration debug command input file
QFORT.BAT	Sample compiler batch file
QFORTLNK.BAT	Sample link batch file
QFORTRUN.BAT	Sample run batch file
FORTRUN.DLL	Dynamic link runtime and debug for IBM Math Co-Processor (only if installed for OS/2 mode and math coprocessor)
FORTRUE.DLL	Dynamic link runtime, debug and emulator for math coprocessor emulation (only if installed for OS/2 mode and emulation)

\APPS\FORTRAN\PLIB (network)
or
\FORTRAN\PLIB (non-network)

This directory is used when linking for OS/2 mode or Family Application Programming Interface applications. It is created only if OS/2 mode libraries are requested. The files in this directory are:

Filename	Contents
FORTRAN.LIB	Objects for intrinsic functions, and imports for runtime and debug (only installed if math coprocessor libraries are requested)
FORTRAE.LIB	Objects for intrinsic functions and imports for runtime, debug, and emulator (only installed if emulation libraries are requested)

\APPS\FORTRAN\RLIB (network)
or
\FORTRAN\RLIB (non-network)

This directory is used when linking for DOS mode or for OS/2 real mode. They are created only if DOS mode libraries are requested. The files in these directories are:

Filename	Contents
FORTRAN.LIB	Objects for intrinsic functions, runtime, debug, and OS/2 to DOS mappings (only installed if math coprocessor libraries are requested)
FORTRAE.LIB	Objects for intrinsic functions, runtime, debug, emulator, and OS/2 to DOS mapping (only installed if emulation libraries are requested)

\APPS\FORTRAN\FLIB (network)
or
\FORTRAN\FLIB (non-network)

This directory is used when linking or binding Family Application Programming Interface applications. They are created only when Family libraries are requested. The files in these directories are:

Filename	Contents
FORTRAN.LIB	Objects for intrinsic functions, runtime, and debug (only installed if math coprocessor libraries are requested)
FORTRAE.LIB	Objects for intrinsic functions, runtime, debug, and emulator (only installed if emulation libraries are requested)

Adding IBM FORTRAN/2 to TopView

Program Information Files are provided for IBM FORTRAN/2 on the IBM FORTRAN/2 "LINK_RUN" master diskette (5¼") or the "INSTALL" master diskette (3½"). These files are required for you to run the programs under TopView. To install IBM FORTRAN/2 under TopView see "Adding Your Programs to TopView" in the *IBM Personal Computer TopView User's Guide*.

When installing the application for TopView, choose the "Add a Program to Menu" option from the TopView Installation Aid menu. Then select "Other" from the list of programs to add.

Note: Do *not* select IBM Professional FORTRAN from the installation menu.

Installing IBM FORTRAN/2 for Use on PC Local Area Network

Install IBM FORTRAN/2 on the network computer using the appropriate installation procedures as described earlier in this chapter. **Warning:** Do not use the Network Installation Aid.

To execute IBM FORTRAN/2 after installation on the network, change the directory to the path selected for the IBM FORTRAN/2 installation, then execute as appropriate for your operating system environment. See "After Installation" on page 2-17.

Editing the Configuration File

After IBM FORTRAN/2 is installed you may need to create or edit the configuration file (CONFIG.SYS). To simplify this procedure, you can concatenate your existing CONFIG.SYS with the CONFIG.DOC file supplied with this product. For example, if you installed your compiler into a directory named \FORTRAN, you could do the following from the root directory:

```
COPY CONFIG.SYS+\FORTRAN\CONFIG.DOC TEMP
```

This creates a file named TEMP that contains your original CONFIG.SYS with the CONFIG.DOC appended.

To preserve your existing CONFIG.SYS, do the following:

```
RENAME CONFIG.SYS CONFIG.BAK
```

then

```
RENAME TEMP CONFIG.SYS
```

For more information on the configuration file, see your operating system manual.

Notes:

1. If the configuration file does not exist on your boot disk or diskette, you must create one. Be sure to name the file CONFIG.SYS.
2. If you create or modify the configuration file, you must reboot your system before you continue. The new configuration remains in effect until you edit CONFIG.SYS and reboot the system again.
3. To edit or create CONFIG.SYS, use an editor such as EDLIN.

Increasing the Number of Open Files Under DOS

With IBM FORTRAN/2, it is recommended that you to be able to open at least 20 files at one time. This increases the number allowed by the DOS configuration default.

To increase the number of files you can open concurrently to 20 or more, add (or modify) the following statement to CONFIG.SYS:

```
FILES=20
```

Increasing the Number of Buffers Under OS/2 and DOS

To improve overall performance, you should increase the number of buffers allocated by OS/2 or DOS at system start up. At least 10 buffers should be allocated.

To increase the buffer allocation, add (or modify) the following statement in CONFIG.SYS:

```
BUFFERS=10
```

Increasing the DOS Mode Size Under OS/2

To reserve enough memory space to run IBM FORTRAN/2 and the debugger, you may need to change the value of RMSIZE in CONFIG.SYS. The minimum recommended value is 640, for example:

```
RMSIZE=640
```

Dynamic Link Libraries Directory under OS/2

Under OS/2, the LIBPATH statement in CONFIG.SYS should include the name of the directory in which the dynamic link libraries are placed. The installation default for a non-network fixed disk is:

```
LIBPATH=C:\FORTRAN
```

For a diskette installation, the default is the root directory. It is specified as:

```
LIBPATH=n:\
```

where *n* is the letter of the drive containing the diskette with the directory.

If the CONFIG.SYS file already contains a LIBPATH statement, add a semicolon (;) and the new path to the existing statement. For example, if CONFIG.SYS contains

```
LIBPATH=C:\
```

then change it to

```
LIBPATH=C:\;C:\FORTRAN
```

The dynamic link libraries for IBM FORTRAN/2 are FORTRUN.DLL and FORTRUE.DLL.

BREAK Command

Under OS/2 and DOS, the automatic interrupt function generates interrupts only during standard input, standard output, standard print, or auxiliary device operations. The function is invoked by simultaneously pressing the CTRL and BREAK keys.

To expand this function so that interrupts are generated during any OS/2 or DOS actions, add the value:

```
BREAK ON
```

to your CONFIG.SYS file.

Sample AUTOEXEC.BAT, CONFIG.SYS, and STARTUP.CMD Files

The following sample files are created during installation as AUTOEXEC.DOC, CONFIG.DOC, and STARTUP.DOC.

The sample statements shown in the following examples are meant to be used as a guide. Paths may be different on your system depending on where your compiler was installed. Sample PATH and LIBPATH statements should be appended to existing PATH or LIBPATH statements. Deleting existing paths can cause unpredictable results.

See the *OS/2 User's Reference* manual or *DOS Reference* manual for more information about how to use these statements.

Sample CONFIG.SYS for DOS

```
FILES=20
BUFFERS=20
BREAK ON
```

Sample CONFIG.SYS for OS/2

```
LIBPATH=c:\FORTRAN
SHELL=COMMAND.COM /e:1000 /p
RMSIZE=640
BUFFERS=20
BREAK ON
```

Sample AUTOEXEC.BAT

```
PATH=C:\FORTRAN
SET LIB=C:\FORTRAN\RLIB
SET FORTRAN.CER=C:\FORTRAN\FORTRAN.CER
SET FORTRAN.DER=C:\FORTRAN\FORTRAN.DER
SET FORTRAN.ERR=C:\FORTRAN\FORTRAN.ERR
SET FORTDBG.HLP=C:\FORTRAN\FORTRAN.HLP
```

Sample STARTUP.CMD or OS2INIT.CMD

```
PATH=C:\FORTRAN
SET LIB=C:\FORTRAN\PLIB
SET FORTRAN.CER=C:\FORTRAN\FORTRAN.CER
SET FORTRAN.DER=C:\FORTRAN\FORTRAN.DER
SET FORTRAN.ERR=C:\FORTRAN\FORTRAN.ERR
SET FORTDBG.HLP=C:\FORTRAN\FORTDBG.HLP
```

Editing the AUTOEXEC.BAT and STARTUP.CMD or OS2INIT.CMD Files

After IBM FORTRAN/2 is installed, you may want to edit the AUTOEXEC.BAT file for DOS mode, or STARTUP.CMD or OS2INIT.CMD file for OS/2 mode.

If you do not currently have a STARTUP.CMD or AUTOEXEC.BAT file in your root directory, and you want your environment set automatically, copy STARTUP.DOC and AUTOEXEC.DOC to the root directory and rename them STARTUP.CMD and AUTOEXEC.BAT respectively.

If you want positive control over the setting of the environment *and* if your IBM FORTRAN/2 files are not in the root of your directory, rename STARTUP.DOC and AUTOEXEC.DOC to STARTUP.CMD and AUTOEXEC.BAT respectively, in the same directory. You should execute one of these, as appropriate, after changing directories to IBM FORTRAN/2. For more information, see your operating system manuals.

PATH, APPEND, and SET Command

You should also add (or modify) the appropriate PATH, APPEND, and SET commands in AUTOEXEC.BAT or STARTUP.CMD. They provide automatic access to IBM FORTRAN/2 when DOS is booted, when DOS mode is entered for the first time after booting, or when a new OS/2 protected mode session is entered.

Sample Batch Files

The following batch files set up your configuration and allow you to compile, link, and run a program.

The Q batch files are supplied as samples. They can be edited to suit specific needs. In particular, names of drives and directories need to be changed to match drives and directories used in the installation.

The batch file QFORT.BAT will compile an IBM FORTRAN/2 program:

```
ECHO OFF
REM COMPILE A FORTRAN PROGRAM
SET FORTRAN.CER=\FORTRAN\FORTRAN.CER
\FORTRAN\FORTRAN %1 %2 %3 %4 %5 %6 %7 %8 %9
SET FORTRAN.CER=
```

The batch file QFORTLNK.BAT will link an IBM FORTRAN/2 program in order to create an OS/2 mode EXE file (change the SET LIB command to reference RLIB instead of PLIB to create a DOS or real mode EXE file):

```
ECHO OFF
REM LINK A FORTRAN PROGRAM
SET LIB=\FORTRAN\PLIB
\FORTRAN\LINK %1 %2 %3 %4 %5 %6 %7 %8 %9
SET LIB=
```

The batch file QFORTRUN.BAT will execute an IBM FORTRAN/2 program:

```
ECHO OFF
REM RUN A FORTRAN PROGRAM
SET FORTRAN.ERR=\FORTRAN\FORTRAN.ERR
SET FORTRAN.DER=\FORTRAN\FORTRAN.DER
SET FORTDBG.HLP=\FORTRAN\FORTBDG.HLP
%1 %2 %3 %4 %5 %6 %7 %8 %9
SET FORTRAN.ERR=
SET FORTRAN.DER=
SET FORTDBG.HLP=
```

Note: You need a library path (LIBPATH in CONFIG.SYS) to FORTRUN.DLL (math coprocessor) or FORTRUE.DLL (emulation) in OS/2 mode.

Refer to your operating system manual if you are not familiar with directories, paths, or batch files.

Compiling the Sample Program

By following the prompts from the installation program and responding as necessary, you should be able to install IBM FORTRAN/2 without any difficulty. Successful installation of IBM FORTRAN/2 can be verified by compiling a sample program called DEMO.FOR. A source file of the program is found on the work diskette or fixed disk directory containing the compiler.

1. If you are using diskettes, insert the "COMPILER" work diskette in drive B. Insert a blank, formatted diskette in drive A.

   Copy DEMO.FOR from the drive containing the compiler to the blank formatted diskette. Issue the command:

   ```
   COPY B:DEMO.FOR A:
   ```

   If you are using a fixed disk, use the installed DEMO.FOR program.

2. For diskettes, enter the following commands:

   ```
   A:
   PATH B:\
   SET LIB=B:\
   SET FORTRAN.CER=B:\FORTRAN.CER
   SET FORTRAN.ERR=B:\FORTRAN.ERR
   ```

For fixed disk, enter:

```
d:
CD path
SET LIB=xlib
```

where *d* is the drive letter of the fixed disk on which the product was installed, *path* is the name of the directory containing DEMO.FOR, and *xlib* is the pathname of the appropriate library.

For the default installation, the directory containing DEMO.FOR is:

```
\APPS\FORTRAN (network)
    or
\FORTRAN (non-network)
```

For the default installation, the name of the library directory is one of the following:

PLIB OS/2 mode
RLIB DOS mode

3. Enter the compile command:

```
FORTRAN DEMO /L
```

or, if you do *not* have a math coprocessor,

```
FORTRAN DEMO /LN
```

The /L compiler option causes the compiler to produce a listing. As the compilation proceeds, the listing is scrolled on your screen. When the compilation is complete, the following message is displayed:

```
Compilation Complete:    0 Errors,    0 Warnings.
```

If you do not receive this message, make sure that you have the required hardware and software and that you have properly installed the compiler.

You can redirect the compiler listing from your screen to the printer to make it available at a later time. To do this, issue the following compile command:

```
FORTRAN DEMO /L >PRN
```

or, if you do *not* have a math coprocessor,

```
FORTRAN DEMO /LN >PRN
```

Refer to the *DOS Reference* manual or the *OS/2 User's Reference* manual for more information about redirecting output.

4. A single file, DEMO.OBJ, should be created on the A drive or the current directory by the compiler. To check this, enter the following command:

   ```
   DIR
   ```

   You should see the following filenames on your screen:

   ```
   DEMO.FOR
   DEMO.OBJ
   ```

   For a fixed disk, you will also see other filenames.

   Now you are ready to link the sample program.

Linking the Sample Program

1. If you are using diskettes, remove the "COMPILER" work diskette from drive B. Insert the work diskette for the appropriate library into drive B as indicated below:

RLIB	1.44MB, 1.2MB, or 720KB diskette for DOS mode.
RLIBN	360KB diskette for DOS mode and math coprocessor.
RLIBE	360KB diskette for DOS mode and emulation.
PLIB	1.44MB, 1.2MB, 720KB, or 360KB diskette for OS/2 mode.

2. Enter the link command:

   ```
   LINK DEMO;
   ```

   If you want to print the link map, use the following link command instead:

   ```
   LINK DEMO,,PRN;
   ```

   Either command creates a load module called DEMO.EXE on the A drive or the current directory.

3. When the link is complete, the DOS or OS/2 prompt is displayed on your screen. Use the DIR command to see that the DEMO.EXE file was created.

   Now you are ready to run the sample program.

Running the Sample Program

1. If you are using diskettes, remove the work diskette containing the library from drive B and insert one of the following:

COMPILER	1.44MB, 1.2MB, or 720KB work diskette
RUNTIME	360KB work diskette

   It contains the runtime message file, FORTRAN.ERR. If you are using a fixed-disk, FORTRAN.ERR is in the same directory as the compiler. Though this file is not required for the sample program, it is a good idea to develop the habit of keeping the FORTRAN.ERR file available at runtime.

   For OS/2 mode, the LIBPATH command in CONFIG.SYS should include the pathname of the directory containing the dynamic link modules for IBM FORTRAN/2. (See "Dynamic Link Libraries Directory under OS/2" on page 2-18.)

2. To run the sample program, enter the name of the load module. You do not have to give the .EXE extension:

   ```
   DEMO
   ```

   The sample program displays the results of two methods for calculating the sine of a number:

   - Using the SIN intrinsic function
   - Using the Taylor series approximation method.

   When the program is complete, the DOS or OS/2 prompt is displayed.

3. If you want to print the results, you can redirect the screen output to the printer by using the following command:

   ```
   DEMO >PRN
   ```

Now you have installed IBM FORTRAN/2 and compiled, linked, and run a sample program. Although you have been introduced to the compilation process, you need to learn about additional compiler options. A full explanation of all compiler options is provided in the next chapter.

If you were unable to compile, link, and run the sample program:

1. Make sure the libraries appropriate to your configuration are installed.
2. Make sure the appropriate work diskettes are used.
3. Make sure that you restarted the system after changing CONFIG.SYS.

Chapter 3. Compiling Your Program

Note: Before you compile your program, you must first set your system environment. See "Editing the Configuration File" on page 2-17 for more information.

For an IBM FORTRAN/2 program to be run on your computer system, it must be compiled and linked. Compilation is performed by the IBM FORTRAN/2 compiler. Its use is explained in this chapter.

The linker is used to link an IBM FORTRAN/2 compiled program. The linker is explained in Chapter 4, "LINK: The Object Linker."

The IBM FORTRAN/2 compiler performs the following actions on the contents of your source file:

- Reads source program units
- Creates an object file by translating the source code into machine-readable object code
- Displays error and warning messages, if any
- Generates a program listing upon request
- Displays status messages during and at the end of compilation.

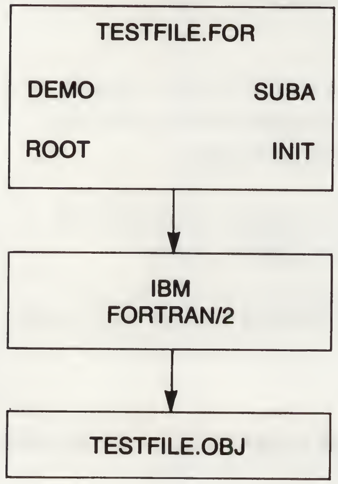
Creating the Source File

Before you compile, you create and name your source file. An IBM FORTRAN/2 source file can be comprised of a number of individual program units. These can be main program units, or subroutine, function, or block data subprogram units.

If you do not supply an extension when issuing the compile command, the IBM FORTRAN/2 compiler expects a file extension of .FOR for your source file. In the following illustration, the single source file TESTFILE.FOR contains four separate program units:

- Main program unit DEMO
- Subprogram unit SUBA
- Function subprogram unit ROOT

- Block data subprogram unit INIT.



When you name your source file, add the extension .FOR to the filename. If you do not, you must specify the correct extension in the FORTRAN compile command. See "FORTRAN Compile Command" on page 3-5.

Creating and Naming Object Files

The IBM FORTRAN/2 compiler creates one object file for each source file. The object file is given the same name as the source file, with the extension .OBJ. In the previous example, the object file is named TESTFILE.OBJ.

The object file is always written to the current directory.

Note: The compiler overwrites an object module when the program being compiled has the same name as one already in the DOS or OS/2 directory.

Starting the Compiler

To start the IBM FORTRAN/2 compiler, use the FORTRAN command and your selected compiler options. But before you start the compiler, you must build the appropriate paths so that DOS or OS/2 can find the compiler software. These procedures are different for diskette and fixed disk systems.

Diskette System

For any 1.44MB, 1.2MB, 720KB, or 360KB diskette drive systems where the source file is on drive *x* and the "COMPILER" work diskette is in drive *y*, enter the following command:

```
x:
```

This command changes your default drive to *x*, which should be the drive containing your source file.

Make the directory containing the source the current directory by using the command:

```
CD pathname
```

where pathname is the name of this directory.

The compiler always writes the object file in the current directory on the default drive.

```
PATH y:
```

This command directs DOS or OS/2 to search the default directory and then the root directory on drive *y* before indicating that a specified .EXE, .COM, .CMD, or .BAT file cannot be found.

```
SET FORTRAN.CER=y:\FORTRAN.CER
```

This command directs DOS or OS/2 to search for the compiler message file on drive y.

Fixed Disk System

If you are compiling a source file that is located on a directory other than that containing the compiler, make the drive with the source the current drive. Issue the command:

```
x:
```

where *x* is the letter of the drive.

Make the directory containing the source the current directory. Issue the command:

```
CD pathname
```

where *pathname* is the name of this directory.

Before compiling, enter the following PATH command:

```
PATH pathname
```

where *pathname* is the name (including drive letter) of the directory containing the compiler (FORTRAN.EXE).

This command tells DOS or OS/2 to always search the specified directory on the default drive for .EXE, .COM, .CMD, and .BAT files.

Also, issue the command:

```
SET FORTRAN.CER=pathname\FORTRAN.CER
```

where *pathname* is the name (including drive letter) of the directory containing the compiler message file.

This command directs DOS or OS/2 to search for the compiler message file in the specified directory.

Note: You may use the APPEND command instead of the SET command:

```
APPEND pathname
```

Purpose

Compiles a source program according to specified options and creates an object module.

Format

FORTRAN *filename options*

filename Is the partially or fully qualified pathname of the source file.

You must specify an extension when the source file has an extension other than .FOR.

Do not include the extension when the source file has the extension .FOR.

The filename can be any valid pathname or filename, according to DOS or OS/2 rules and conventions. The object module will always be written to file *filename* with the extension changed to .OBJ in the current directory.

options Are any of the available compiler options listed below. Compiler options can appear in any order, and should be preceded by a slash (/). Only one / is required for multiple options on the same command line, except in the case of the /C and /P options. They require values, and the next option following the value must be preceded by a /. If you separate options with spaces, a slash must precede each option.

After you enter the compile command, the compiler displays the message:

```
IBM FORTRAN/2 Compiler
Version 1.00
(C)Copyright IBM Corp 1984, 1987
(C)Copyright Ryan-McFarland Corp 1984, 1987
```

Note: For a diskette system, insert the "COMPILER" work diskette containing FORTRAN.EXE before you compile.

Compiler Options

Compiler options control and modify your program's compilation. They can:

- Generate different types of listing information
- Modify the appearance of your listing
- Specify the default method for processing integer data
- Specify the size of assumed-size, adjustable, and dummy arrays
- Insert extra code and data for the IBM FORTRAN/2 Interactive Symbolic Debug program
- Determine whether DO-loops and character constants are handled according to the FORTRAN 66 or FORTRAN 77 standard
- Determine whether the program is portable under System Application Architecture (SAA) Guidelines. See page 3-11 for a description of SAA.
- Create processor-specific code
- Suppress code optimization.
- Suppress code generation (syntax check only)
- Conditionally compile debug lines
- Create code that does not depend on the presence of the math coprocessor.

No physical limits are placed on the number of compiler options you can specify during a single compilation. However, because certain compiler option combinations are incompatible, logical restrictions occur (as mentioned in the *options* list).

Option	**Action**
/A	Adds a listing of the program allocation map to the standard listing. This option must be used in concert with the /L option. The default is to not add the map to the standard listing.

/B Assumes all adjustable and assumed-size arrays are larger than 64KB. The addressing structure of the microprocessor accesses arrays that are less than or equal to 64KB more efficiently than those larger than 64KB.

Because the sizes of the adjustable or assumed-size arrays are not known until the program runs, the /B option generates code capable of accessing adjustable and assumed-size arrays that are larger than 64KB.

The default (if the /H option is not specified) assumes all adjustable and assumed-size arrays are less than 64KB.

See "Interaction Between the /B and /H Options" on page 3-12 for additional information.

Note: This option only applies to subprograms. Main programs cannot have adjustable or assumed-size arrays, and the /B option is ignored. If you create a library of general-purpose subroutines, be sure that any subroutines that might be required to process arrays greater than 64KB are compiled with the /B option. Since additional overhead is required to process large arrays, you might want to create two libraries.

/C *n* Sets the line length of the program listing to the number specified by *n*. This number is the number of columns per line. Lines longer than this number are truncated. The number *n* ranges from 1 to 132. The /C and the n can be separated by optional spaces.

The default is 80 columns per line.

/D Compiles debug source code lines (those with a "D" or "d" in column one).

The default treats these lines as comments.

/E Suppresses the source code listing but shows error and warning messages in the program listing. This option implies the /L option.

The default for /L alone generates a program listing containing all messages as well as the source code.

/F Directs that all DO-loops be performed at least once before the value of the DO variable is tested (the FORTRAN 66 standard) and treats character constants as Hollerith (also FORTRAN 66).

The default is to test the value of the DO variable before performing the DO-loop (the FORTRAN 77 standard) and to treat character constants as character type only.

/H Instructs the compiler to assume that any dummy array whose size is less than 64KB is contained within more than one segment.

The addressing structure of your system is designed to access single segment arrays more efficiently than those contained in two or more segments. The compiler knows if local or common arrays are single or multi-segment arrays and generates the appropriate object code. However, in the case of dummy arrays this is not known until program execution. Selecting /H directs the compiler to generate code capable of accessing multi-segment arrays for all dummy arrays in the compilation. This option does not affect any other type of array.

The default is to not make this assumption.

See "Interaction Between the /B and /H Options" on page 3-12 for additional information.

Note: This option only applies to subprograms. Main programs cannot have dummy arrays, and the /H option is ignored. If you create a library of general-purpose subroutines, be sure that any subroutines that might be required to process arrays contained within more than one segment are compiled with the /H option. Since additional overhead is required to handle such dummy arrays, you might want to create two libraries.

/I Allocates 16 bits for all INTEGER data. This makes all INTEGER data perform as INTEGER*2 data. This compiler option affects any data specified as INTEGER. This option does not affect data specified as INTEGER*2 or INTEGER*4. For more information about the effect /I has on integer constants and intrinsic functions that return a result of type integer, see "/I Compiler Option" in Chapter 4, "Portability, Conversion, and Extensions," in the *IBM FORTRAN/2 Language Reference* manual.

An important use of this option is to improve runtime performance and decrease program size.

This option may violate the ANSI X3.9-1978 FORTRAN 77 standard for INTEGER data contained in COMMON or EQUIVALENCE statements.

The default allocates 32 bits for INTEGER data.

/J Generates a page break between sections of the program listing. This option must be used in concert with the /L option.

The default is to generate three blank lines between listing sections.

/L Generates a standard program listing. (See "Compiler Listing" on page 3-15 for details of the listing format.) You must use the /L or /E option if you use the /A, /J, /M, /S, or /X options.

The default does not generate a program listing.

/M Generates an object code listing. This option must be used in concert with the /L option, and is ignored if the /U option is present.

The default does not generate this part of the program listing.

/N Creates code that does not depend on the presence of the math coprocessor. If your machine does not have a math coprocessor you should select this option. Code generated with this option will use the math coprocessor if present.

The default is to create code that uses the math coprocessor.

See "Running Programs Compiled With /N" on page 5-10 for additional information.

/P *n* — Sets the page length of your program listing to the number *n*. This number is the number of lines per page minus 1. The *n* entry and the option can be separated by an optional space.

Note: *n* must be greater than 4.

The default is 64, meaning that 63 lines are printed per page.

/Q — Suppresses the display of individual compilation summaries on the standard error device.

The default is to generate individual program unit summaries.

/S — Adds summary information to the standard program listing. This option must be used in concert with the /L option.

The default is to not add summary information.

/T — Creates object code that runs under control of the IBM FORTRAN/2 Interactive Symbolic Debug program (called Debug). This option implies the /Z option. See Chapter 7, "Debugging Your Program" for more information.

The default does not compile the program for Debug.

/U — Suppresses the generation of code and the object file. This provides a faster way of finding errors and warnings in a source file.

/V — Directs the compiler to perform Federal Information Processing Standard flagging. All FORTRAN language features which are extensions to the ANSI X3.9-1978 FORTRAN 77 standard are flagged.

Note: If you use the /W option with the /V (or /V1) option, you are suppressing warnings which indicate

incorrect or possibly incorrect code. Therefore, an error-free compilation does not necessarily indicate a standard-conforming program. It is recommended that the /W option not be used with the /V (or /V1) option.

/V1 Directs the compiler to perform System Application Architecture flagging. All FORTRAN language features which are not portable according to the SAA guidelines are flagged. A space may *not* occur between V and 1.

The System Application Architecture guidelines define a portable FORTRAN program as one that only uses features from the following list:

ANSI X3.9-1978 FORTRAN 77 standard
INTEGER*2 types and type statements
INTEGER*4 types and type statements
LOGICAL*1 types and type statements
LOGICAL*4 types and type statements
REAL*4 types and type statements
REAL*8 types and type statements
COMPLEX*8 types and type statements
COMPLEX*16 types and type statements
Type statements with data initialization
IMPLICIT NONE statement
Mixed expressions allowing double precision and complex values
Format descriptor "Z" for hexadecimal
Upper/lower case insensitivity
ISA S61.1 Bit Manipulation Intrinsic functions
COMPLEX*16 intrinsic functions
INCLUDE statement
Names longer than 6 characters
Non-leading underscore characters in names
Association of character and noncharacter items in EQUIVALENCE and COMMON statements
HFIX intrinsic functions

/W Suppresses warning messages (but not error messages).

The default generates warnings.

/X Generates a cross-reference listing of identifiers and labels in the program listing. This option must be used in concert with the /L option.

The default does not generate a cross-reference listing.

/Y Creates 80286/80287 specific code.

/Y1 Creates DOS mode specific code.

/Y2 Creates 80286/80287 DOS mode specific code.

/Y3 Creates 80286/80287 OS/2 mode specific code.

Notes:

1. For compiler options Y1, Y2, and Y3, a space may *not* occur between the Y and the following digit.
2. Only one of the /Y or /Yn options should be specified.

The default is to generate code that will execute correctly on all supported computer systems in both protected and real mode.

/Z Suppresses code optimization. The default performs code optimization. See Appendix C, "Code Optimization" in the *IBM FORTRAN/2 Language Reference* manual for information about code optimization.

Interaction Between the /B and /H Options

The table that follows illustrates the effects of the /B and /H options, singly or in concert, on the number of segments assumed for a dummy array. In using the table, keep in mind that the /B option applies to adjustable and assumed-size dummy arrays only, whereas the /H option applies to all dummy arrays.

Note that, for the purposes of the table:

- *n* indicates multiple segments and 1 means contained in one segment.
- *Less than 64KB* means a dummy array that is known at compile time to have a size <= 65536 bytes (and, therefore, it is not an adjustable or assumed-size dummy array).

- *Unknown* means the array is an adjustable or assumed-size dummy array. That is, its size is unknown at compile time.
- *Greater than 64KB* means a dummy array that is known at compile time to have a size > 65536 bytes (and, therefore, it is not an adjustable or assumed-size dummy array).

Compiler Option \ Memory Size of Dummy Array Segment	Less Than 64KB	Greater Than 64KB	Unknown
No Option	1	n	1
/B	1	n	n
/H	n	n	n
/H and /B	n	n	n

Figure 3-1. The Effect of the /H and /B Options on Dummy Array Segments

Compile Command Errors

When you specify a file (any valid pathname or filename) that does not exist in the current (or specified) drive or directory, the compiler displays the message:

```
[00002] OPEN Error on File:  Filename
```

To correct this problem, make sure you have entered the correct file name and extension (when appropriate). Check that the file is resident on the proper drive and the current directory.

When you request an option that does not exist or you have not followed the syntax rules correctly, the compiler displays the following message:

```
Invalid option specified
Usage: FORTRAN pathname [/ABDEFHIJLMNQSTUWXZ] [/V[1]] [/Y[1|2|3]]
                        [/C n] [/P n]
```

FORTRAN
Compile Command

Check the options list within this message to make sure you have entered valid compiler options and that you have entered them correctly.

The following are examples of valid and invalid compile commands:

Valid Compile Commands

```
FORTRAN TESTFILE
```

This command compiles the source file TESTFILE.FOR and creates the appropriate object file. No options are requested.

```
FORTRAN FPC\FORTRAN\TESTFILE /L
```

This command compiles the source file FPC%FORTRAN%TESTFILE.FOR; creates an object file (in the current directory); and produces a standard listing (/L) of the compiled source program.

```
FORTRAN TESTFILE.ABC /LC 65/X
```

This command compiles the source file TESTFILE.ABC; creates an object file; writes a program listing (/L) of 65 characters per line (C 65); and produces a cross-reference listing of each label and identifier found in the compiled program (/X). Note that only one slash was required before the /L and /C options. A second slash precedes the /X option, because it follows the /C option and its required value.

Invalid Compile Commands

```
FORTRAN
```

This command is invalid because it does not specify a source file, however, it displays a message explaining the proper usage.

```
FORTRAN TESTFILE.ABC /P 33 X
```

This command is invalid because the option following the /P option must be preceded by a slash.

Compiler Listing

When you request the /L compiler option, a listing of your program is displayed on the DOS or OS/2 standard output device (your screen). You can redirect this listing to a disk file by using the following compile command:

```
FORTRAN fname /L >fname.LST
```

or to the printer using the following command:

```
FORTRAN fname /L >LPT1
```

Depending on the options you select, your listing will contain some or all of the following information:

Option	Program Listing Section
/L	Listing Header
/L	Source Code and Statement Diagnostics
/LA	The Allocation Map
/L	Program Unit Diagnostics and Allocation Diagnostics
/LX	Cross-Reference Listing
/LM	Object Code Listing
/LS	The Called Subprogram Summary
/LS	The Statement Label Summary
/LS	The Statement Location Summary
/LS	The Program Unit Summary
/L	Compilation Summary

Note: If /E is specified, the /L option is not required.

Listing Header

The Listing Header appears at the top of each page and contains the following information:

- Compiler version
- Page number
- First 21 characters of the compiled source filename
- Date and 24-hour time the listing was run
- First 16 characters of the compiler options selected.

The header looks like this:

```
IBM FORTRAN/2 Compiler  V1.00      Page          1
Source File: DEMO.FOR      Options: /lsax      01/08/87  10:49:55
```

Source Code and Statement Diagnostics

This section contains the following information for each source line in the program:

- Line number of the source line
- Label (if any) of the source statement
- Source line
- Statement diagnostic marker and message, if any.

The following is a sample of this section of the listing:

```
   1        FUNCTION ROOT (A,Q)
   2        COMMON /COM1/ B,C,D,E
   3       1       /COM2/ X,Y
   4        X=1.2
   5        Y=A*3.14159
   6        DO 21 I=1,10
            •
            •
            •
  26        END
```

When your source line contains a statement error (such as improper syntax), it is marked on the listing with a question mark (?). This mark appears directly underneath the invalid character or name. A message appears on the next line after the mark explaining the nature of the diagnostic and indicating whether the diagnostic is an error or a warning.

When your source code has more than one diagnostic for a single line, all errors in the line are marked. The associated messages are numbered and displayed beneath the marks in order of their appearance within the source line.

Introducing errors into the sample source code shown above will generate Statement Diagnostics.

Example:

```
   1       FUNCTION JOE (A,Q)
   2       COMMON /COM1/ B,C,A,B
                               ? ?
1 ** ERROR ØØ8: CONFLICTS WITH PREVIOUS DECLARATION OF THIS IDENTIFIER
2 ** ERROR ØØ8: CONFLICTS WITH PREVIOUS DECLARATION OF THIS IDENTIFIER
   3     1      /COM2/ X,X
                           ?
1 ** ERROR ØØ8: CONFLICTS WITH PREVIOUS DECLARATION OF THIS IDENTIFIER
    PREVIOUS ERROR WAS AT LINE    2
   4       X=1.2
   5       y=A*3.14159
   6       DO 21 I=1,1Ø
   7       Z=X*Y
   8       END
             ?
1 ** ERROR Ø48: OPEN DO LOOP
    PREVIOUS ERROR WAS AT LINE    3
```

In this example, line 2 contains two conflicts with previous declarations of the identifier. The first indicates that A has already been defined as a dummy argument (in line 1 above), and the second indicates that B has already been defined (in the same line). Line 3 contains a doubly-defined variable. Also, line 8 indicates that there is an open DO loop somewhere in the program unit.

See Appendix A, "Messages" in the *IBM FORTRAN/2 Language Reference* manual for a complete listing of diagnostic messages.

Diagnostic Threading

IBM FORTRAN/2 provides diagnostic threading facilities. By reading the PREVIOUS ERROR WAS AT LINE message (as shown twice in the extract above), you can track back through every error encountered during compilation. This message appears for every erroneous statement, except the first. A variant of this message — LAST ERROR WAS AT LINE — also appears as part of the compilation summary, and points to the last error in the program.

Compilation always proceeds to the end of the program regardless of the number of diagnostics found, unless an error causes abnormal termination. Global diagnostics, such as illegal control transfers, are listed before the allocation map.

Allocation Map and Diagnostics

This section contains the following information:

- Program unit diagnostics
- Common block allocation
- Dummy argument allocation
- Scalar allocation
- Equivalence allocation
- Array allocation
- Allocation diagnostics.

The compiler lists the following information for each type of allocation:

- Title, indicating the type of allocation
- Location offset, relative to either the local data area or the start of a common block (depending on the allocation)
- Number of bytes (in decimal notation) allocated to the variable
- Class of variable: scalar or array.

  The number of dimensions in the array is also listed. The dimensions are listed in the form *n*D, where *n* is the number of dimensions.
- Type of variable
- Variable name.

Following are examples of Program Unit Diagnostics that would appear before the Allocation Map:

```
** WARNING 122: FUNCTION NAME AND ENTRY NAMES NOT ASSIGNED A VALUE (JOE)
** ERROR Ø48: OPEN DO LOOP (21) AT LINE    6
** ERROR Ø63: UNDEFINED LABEL (21) AT LINE    6
```

The Allocation Map looks like this:

```
COMMON BLOCK/COM1/ ALLOCATION ØØØØØØ2C BYTES

LOCATION    BYTES     CLASS        TYPE          NAME

CØØ-ØØØØ        4     SCALAR       REAL          RC
CØØ-ØØØ4        8     SCALAR       DOUBLE        R8C
CØØ-ØØØC       32     ARRAY 2D     COMPLEX       COMPC

DUMMY ARGUMENT ALLOCATION

LOCATION    BYTES     CLASS        TYPE          NAME

S-ØØØØ          4     SCALAR       REAL          D1
S-ØØØ4          4     SCALAR       LOGICAL*4     D2
S-ØØØ8          4     SCALAR       COMPLEX       D3

SCALAR ALLOCATION

LOCATION    BYTES     CLASS        TYPE          NAME

DØØ-ØØ12        8     SCALAR       DOUBLE        R81
DØØ-ØØ1A        4     SCALAR       LOGICAL*4     COMP1
DØØ-ØØ1E        4     SCALAR       INTEGER*4     JOE
```

```
EQUIVALENCE ALLOCATION

LOCATION    BYTES    CLASS       TYPE         NAME

DØØ-ØØ2A        4    SCALAR      LOGICAL*4    LOG1
DØØ-ØØ2A       4Ø    ARRAY 1D    LOGICAL*4    LOG2
DØØ-ØØ5Ø        2    SCALAR      INTEGER*2    I41

ARRAY ALLOCATION

LOCATION    BYTES    CLASS       TYPE         NAME

DØØ-ØØ56       32    ARRAY 3D    REAL         R42
```

The Allocation Diagnostics appear on a separate page following the Allocation Map.

Example:

```
ERRORS FOUND DURING ALLOCATION

** ERROR Ø56:  SUBSCRIPT USED WITH UNDIMENSIONED VARIABLE (B)
```

See Appendix A, "Messages" in the *IBM FORTRAN/2 Language Reference* manual for a complete listing of diagnostic messages.

Cross-Reference Map

When you select the /X option with /L or /E, the compiler generates a cross-reference map. This contains a list of all labels and symbolic names in your program, along with the line numbers where they appear.

The listing is generated in ascending numeric and alphabetic order.

A slash (/) follows the line number when the reference is a specification statement or defines a statement label. An asterisk follows the line number when the reference is a possible modification.

Example:

```
CROSS REFERENCE

21                  6
A                   1/      5
B                   2/
C                   2/
COM1                2/
COM2                2/
I                   6*
JOE                 1/
Q                   1/
X                   2/      4*      7
Y                   5*      7
Z                   7*
```

Object Code and Code Generation Diagnostics

When you select the /M option with /L or /E, the compiler generates an object code listing. It contains the following information for each statement in the object module:

- Sequential line number of your source line
- Source statement label (if any)
- Source line
- Section types: D for data; T for text (code); Z for constants in code; S for stack; X for debug data; Y for character temporaries; for common blocks, the name of the common block is given.
- Starting hexadecimal address of the assembly instruction
- Instruction's operands (in hexadecimal)
- Notation (R) indicating the operand address is relocatable
- Any compiler-generated statement labels
- Assembly instruction
- Assembly instruction's operands.

The following is an example of this section of the listing:

```
14        PRINT '(10X,''X'',10X,''MYSIN(X)'',11X,''SIN(X)'',11X,
15      *       ''TERMS'',/)'
     D00-0105   4500                    DW      H'0045'
     D00-0107   C0000000        R       DD      D00+H'00C0'
     D00-010B   0E00                    DW      H'000E'
     D00-010D   05010000        R       DD      D00+H'0105'
       T-0019   BB  0B01        R       MOV     BX,OFFSET(D00+H'010B')
       T-001C   8EC2                    MOV     ES,DX
       T-001E   B9  0200                MOV     CX,H'0002'
       T-0021   9A  00000000    R       FCALL   F@IOWEF2
       T-0026   9A  00000000    R       FCALL   F@IOSPF
16        DO 10 I = 1, 10
       T-002B   C706  14000100  R       MOV     WORD PTR LOCAL DATA+H'0014',H'0001'
       T-0031   C706  16000000  R       MOV     WORD PTR LOCAL DATA+H'0026',H'0000'
       T-0037   C746  FA0A00            MOV     WORD PTR [BP+H'FFFA'],H'000A'
       T-003C   C746  F40C00            MOV     WORD PTR [BP+H'FFFC'],H'0004'
       T-0041                      M0002
```

After the object code listing, the following Code Generation Diagnostic can appear:

**ERROR 074: CODE SEGMENT GREATER THAN 64K AT LINE *line-number*

This means that the size of the code segment generated for the program unit has exceeded its maximum value of 64KB (65,536 bytes).

Called Subprogram Summary

This section contains the following information for each called subprogram:

- Class of subprogram: RUNTIME, SUBROUTINE, or FUNCTION
- Number of arguments contained in the subprogram
- Type of function subprogram (when applicable), such as REAL or INTEGER
- Names of all called subprograms within the program unit.

The following is a sample of this section of the listing:

```
CALLED SUBPROGRAMS

CLASS       ARGS   TYPE   NAME

FUNCTION       2   REAL   MYSIN
RUNTIME                   F@IPGM
RUNTIME                   F@IOWEF2
RUNTIME            REAL   F@FSN3
RUNTIME                   F@IORSF
RUNTIME                   F@IOSPF
```

Statement Label Summary

This section contains the following information for all statement labels:

- Location, in hexadecimal, of the labels. The location is relative to the applicable segment of the program.
- Use of the labels within the program.
- Names of all statement labels.

The following is an example of this section of the listing:

```
STATEMENT LABELS

LABEL    LOCATION    USE      LABEL    LOCATION    USE

1Ø       T-ØØ93      DO END   2Ø       DØØ-Ø126    FORMAT
```

Statement Location Summary

This section contains the following information:

- Starting line number of each executable statement
- Starting location, in hexadecimal, of each executable statement.

The following is a sample of this section:

```
STATEMENT LOCATIONS

LINE  LOCATION  LINE  LOCATION  LINE  LOCATION  LINE  LOCATION

   1  0000         4  0028         5  0034         6  0048
   7  0050         8  005A
```

Program Summary Messages

This section contains the size and number of errors and warnings issued for a program unit. In addition, summary information is provided for the entire compilation.

The following is a sample:

```
PROGRAM UNIT SIZE

CODE    DATA        STACK     TOTAL
0065    00000032    000E      000000A5

NUMBER OF WARNINGS  IN PROGRAM UNIT:  0
NUMBER OF ERRORS    IN PROGRAM UNIT:  7
```

Compilation Summary

After all program units within your source file have been compiled, summary information about the compilation as a whole appears.

For instance:

```
     LAST ERROR WAS AT LINE 4

     NUMBER OF WARNINGS  IN COMPILATION:  0
     NUMBER OF ERRORS    IN COMPILATION:  9
```

The LAST ERROR AT LINE *line-number* message is omitted if there are no errors or warnings in the compilation.

Compiler Status Messages

IBM FORTRAN/2 also generates status messages as program units are being compiled. These messages indicate the names of the program units being compiled and the number of errors and warnings encountered during compilation. Unlike the program listing, these messages are written to the standard error device (the screen). These messages are not affected by redirection. If you do not want this information listed to standard error output, enter the /Q option in the compile command.

The following status message is displayed when the compilation of a program unit is started:

```
Compiling program unit
```

where *program unit* is the name of the program unit to be compiled.

The following status message is displayed when the compilation of a program unit is completed:

```
ee Errors, ww Warnings.
```

where:

ee is the number of errors found in the program unit.

ww is the number of warnings found in the program unit.

Compiler Completion Messages

The compiler displays a number of messages to indicate whether it has completed compilation normally or abnormally. One of these messages, Compilation Complete, is displayed each time your compilation completes. This message appears:

```
Compilation Complete: eeee Errors, wwww Warnings.
```

The number of errors, *eeee*, and warnings, *wwww*, are listed.

If you do not receive this message, make sure that you have the required hardware and software, and that you have installed the compiler properly. If you still have problems, it is possible that you need to check your math coprocessor or switch settings in your machine. See your IBM Personal Computer *Guide to Operations* manual for more information.

Abnormal completion messages are displayed under specific circumstances and are listed in Appendix A, "Messages" in the *IBM FORTRAN/2 Language Reference* manual.

Unlike the program listing, compiler completion messages cannot be redirected to a file or device other than your screen.

Unlike the compiler status messages, the compiler completion messages cannot be suppressed.

Chapter 4. LINK: The Object Linker

Introduction

Note: Before you link your program, you must first set up your system environment. See "Editing the Configuration File" on page 2-17 for more information.

The linker produces executable program files or dynamic link libraries from object files. The linker can be used with either OS/2 or DOS.

Executable program files can be produced for OS/2 mode, or for DOS mode.

Note: Where "OS/2 mode" is used in this chapter, this means "OS/2 protected mode." Where "DOS mode" is used in this chapter, it means "DOS and OS/2 real mode."

Linker output, whether you are linking an application or a group of dynamic link routines, is one file. If you link an application, the resulting .EXE file is called an *application module*. If you are linking a set of dynamic link routines, the resulting .DLL file is called a *dynamic link library*.

The default file extension LINK gives to program modules is .EXE. However, when creating a dynamic link library, the default file extension given by LINK is .DLL.

You create an application module by linking your compiled source files with the LINK utility. LINK takes the compiled source files, a list of libraries, and an optional module-definition file and creates an application module.

```
                  +--------------------+
*.OBJ files ----->|        LINK        |-----> .EXE file
*.LIB files ----->|                    | (application module)
                  +--------------------+
                            ^
                            |
                   Module-Definition
                    File (optional)
```

Linking Under Different Operating Systems

There are two operating system environments you can use to link and run programs. LINK creates files to run in OS/2 mode if one or both of the following occur:

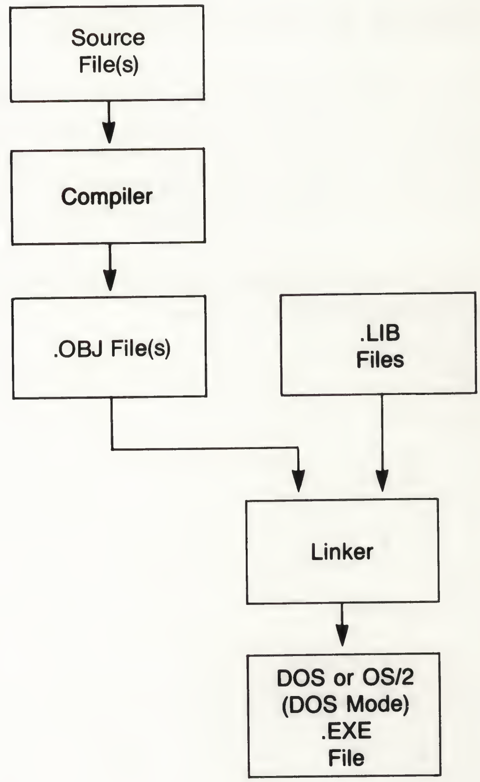
- Dynamic links — If a dynamic link entry resolves any external reference, LINK produces an OS/2 executable file. Although you can produce an OS/2 executable file without dynamic links, you cannot run such a program.
- Definition files — If you request a module-definition file, LINK produces an OS/2 executable file, or a dynamic link library.

A DOS mode executable file is produced if no dynamic link entries and module definition files are present at link time.

Linking for DOS Applications

You create an application module (.EXE file) by linking your compiled source files (.OBJ files) and the appropriate libraries (.LIB files) using the LINK utility.

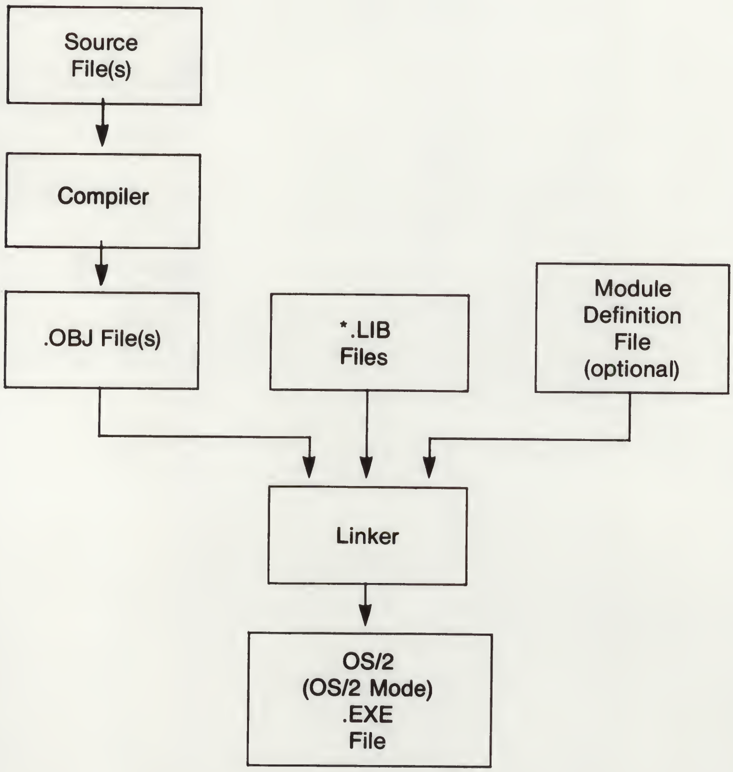
The following diagram illustrates the steps required to create a DOS mode application .EXE file.


Source
File(s)
Compiler
.OBJ File(s)
.LIB
Files
Linker
DOS or OS/2
(DOS Mode)
.EXE
File

Linking for OS/2 Applications

You create an application module by linking your compiled source files (.OBJ files), the appropriate libraries (.LIB files) and an optional module-definition file using the LINK utility. A *module-definition file* is an ASCII text file containing information about your application. LINK uses this information to help build the .EXE file.

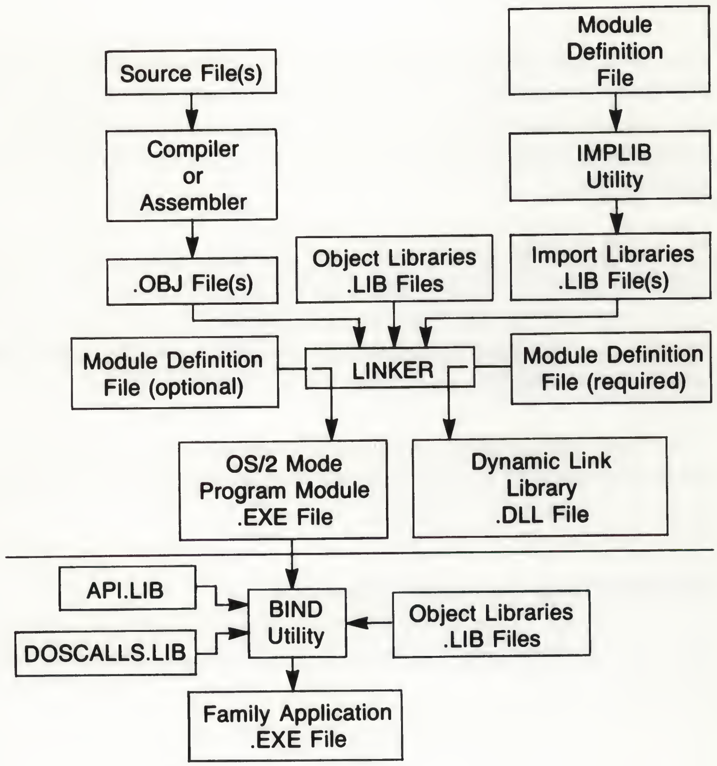
The following diagram illustrates the steps required to create an OS/2 mode application .EXE file:



Linking for Family Applications

You can also produce executable files that can run in either real (compatibility) mode or OS/2 mode. A program that can run in both DOS mode and OS/2 mode is called a *Family Application*. To create a Family Application, use the BIND utility and a mapping library (Application Programming Interface). For more information about BIND, see "Creating Family Applications" on page 4-57, and the *OS/2 Programmer's Guide*.

The following diagram illustrates the steps required to create a Family Application .EXE file:



Using the Linker

You can provide input to the linker in three different ways:

- Answer a series of prompts
- Enter a command line to OS/2 or DOS
- Use a response file.

You can also mix these three methods.

You should allow the linker to prompt you for responses until you feel comfortable with its commands and operations. When you understand the linker prompts and operations, you can then use either of the other methods to run it.

The command line method lets you type all commands, options, and filenames on the line used to start the linker. With the response file method, you can create a file that contains all the necessary commands, options, and filenames and then tell the linker where to find that file.

To use the linker, create and submit one or more object files along with any required library files. The linker combines the code and data in the object files and searches the library files that you name to resolve external references to routines and variables. The linker then writes the executable (.EXE) file.

The linker can produce executable files containing up to one megabyte of code and data for DOS (DOS mode), or 16 megabytes of code and data for OS/2 mode.

Terminating the LINK Session

You can stop the linker at any time by pressing the CTRL-C or CTRL-BREAK keys.

Options

Select linker options by typing them after the filename at any linker prompt. See "Using LINK Options" on page 4-20.

Using Prompts to Specify LINK Files

Start the linker by typing LINK at the OS/2 or DOS prompt. The linker prompts you for the information it needs by displaying the following lines, one at a time. The linker waits for you to respond to each prompt before displaying the next prompt.

Object Modules [.OBJ]:
Run File [filename.EXE]:
List File [NUL.MAP]:
Libraries [.LIB]:
Definitions File [NUL.DEF]:

You can use any combination of uppercase and lowercase letters for the filenames you give in response to the prompts. For example, the following three filenames are equivalent:

```
abcde.fgh
AbCdE.FgH
ABCDE.fgh
```

To enter a filename that has no extension, type the name followed by a period. For example, if you type **ABC.** in response to a prompt, it tells the linker that the given file has no extension. On the other hand, typing only **ABC** tells the linker to use the default extension for that response.

At the **Object Modules** prompt, list the names of the object files you want to link. You must respond to this prompt. There is no default.

The linker automatically supplies the .OBJ extension when you give a filename without an extension. If your object file has a different extension, you must supply it for the file to be found.

You can use pathnames with object filenames. You can give the linker the pathname of an object file in another directory or on another diskette. If the linker cannot find a given object file, it displays a message and waits for you to change diskettes.

You must separate each object filename from the next by blanks (spaces) or a plus sign (+). If a plus sign is the last character typed on the line, the **Object Modules** prompt reappears on the next line, allowing you to name more object files.

Do not embed a plus sign in a filename entry; use the plus sign after a complete filename only.

Here is a sample session using the separated entries method of entering a filename:

```
Object Modules [.OBJ]: SHAPE+SQUARE+TRIANGLE+ (Press Enter)
Object Modules [.OBJ]: RECTANGLE+ELLIPSE+POLYGON+ (Press Enter)
Object Modules [.OBJ]: VECTOR (Press Enter)
```

After you give all object filenames, press Enter. The linker displays the following prompt:

Run File [*filename*.EXE]:

The *filename* in this prompt is the same as the first one that you entered in response to the **Object Modules** prompt. If you want the linker to supply a default executable-filename, press Enter. The linker gives the executable file the same filename as the first object file, but adds the extension .EXE (as shown in the prompt). You can give any filename you wish. However, you should use an .EXE extension, because OS/2 and DOS run files with this extension.

When responding to linker prompts, the default **Run File** extension displayed is **.EXE**. If the user is creating a dynamic link library, the linker actually creates a file with the extension **.DLL** and issues a message.

The linker displays the prompt:

List File [NUL.MAP]:

Enter the name of the map (.MAP) file that you want to create. A .MAP file contains the names of all segments in the order of their

appearance in the load module. By adding the /MAP option you can also list all external (public) symbols and their addresses. See "The Map File" on page 4-83 for more information.

If you have a filename without an extension, the linker provides the .MAP extension. The linker creates the .MAP file in the current working directory, unless you specify a different path.

You can skip this prompt by pressing Enter without giving a name. When you skip this prompt, the linker uses the special filename NUL.MAP, which tells the linker not to create a listing file.

The linker displays the prompt:

Libraries [.LIB]:

Following the **Libraries** prompt, you can specify nothing, one entry, or more than one entry, separated by spaces or a plus sign. If the plus sign is the last character typed, the **Libraries** prompt reappears on the next line, allowing you to type in additional entries. Each entry can be either a directory specification or a library name. Directory specifications must end with a backslash (\) so that the linker can distinguish the directory names from the library names.

If you do not wish to search any libraries other than default libraries, do not enter any names. Just press Enter.

When you give directory specifications, the linker uses them to search for the default libraries and for any other libraries which do not have a pathname on the same line. To locate the default libraries, the linker searches in the following order:

1. The current working directory
2. The directories in the order listed following the **Libraries** prompt
3. The libraries specified by the LIB environment variable.

When you specify a library name, the linker searches for the library and resolves external references. If the library name includes a directory specification, the linker searches only that directory for the library. If you give no directory specification, the linker searches for the library in the order just described. LINK searches all libraries until it finds the first definition of a symbol.

Unless you supply an extension, the linker automatically supplies the .LIB extension.

For IBM FORTRAN/2 object files, the default library is FORTRAN.LIB if /N was not specified in the compile, or FORTRAE.LIB if /N was specified. You must specify the appropriate directory or use the appropriate diskette containing the library corresponding to whether you want an OS/2 mode .EXE or a DOS mode .EXE file. The directory can be specified in the response to the LIBRARIES prompt or in LIB variable set by the SET command.

If you do not want to link with the default IBM FORTRAN/2 library, you can give the name of a different library. In this case, use the /NOD option. See "/NODEFAULTLIBRARYSEARCH" on page 4-31 for more information.

The linker displays the prompt:

Definitions File [.DEF]:

The **Definitions File** prompt lets you supply the name of a module-definitions file if you are using one with the program being linked. The linker allows you to supply any name and extension for a definitions file. If you give no extension, the linker searches for the default extension .DEF.

A module-definition file is for advanced OS/2 applications and dynamic link libraries. It is not required for a link of IBM FORTRAN/2 objects for an application module. See "Module Definition Files" on page 4-62 for more information.

You can skip this prompt by pressing Enter without giving a name. When you skip this prompt, the filename NUL.DEF is used. It tells the linker not to search for a definitions file.

When you enter filenames, you must give a path name for any file that is not on the current drive or in the current directory.

Example Using Prompts

The following example links the object modules MODA.OBJ, MODB.OBJ, MODC.OBJ, and STARTUP.OBJ. LINK searches the library file MATH.LIB in the \LIB directory on drive B for routines and data used in the programs. LINK then creates an executable file named MODA.EXE and a map file named ABC.MAP. The /PAUSE option in the **Object Modules** prompt line causes LINK to pause while you change diskettes. LINK then creates the executable file.

```
LINK

Object Modules [.OBJ]: moda + modb +
Object Modules [.OBJ]: modc + startup /PAUSE
Run File [MODA.EXE]:
List File [NUL.MAP]:abc
Libraries [.LIB]:  b:\lib\math
Definitions File [NUL.DEF]:
```

Selecting Default Responses

To select the default response to a prompt, press Enter without typing a file name. The next prompt appears.

Use the semicolon character (;) to save time when the default responses are acceptable. LINK does not allow the semicolon character with the first prompt, **Object Modules [.OBJ]:**, because that prompt has no default.

To select all the remaining default responses at once, type a semicolon (;) at the next prompt and press Enter. When you use the semicolon, you cannot respond to any of the remaining prompts for that link session.

Defaults for the other linker prompts are as follows:

- For the **Run File** prompt, the default is the name of the first object file specified for the previous prompt, and LINK replaces the .OBJ extension with the .EXE or .DLL extension.
- For the **List File** prompt, the default is the special file name NUL.MAP, which tells LINK not to create a map file.
- For the **Libraries** prompt see page 4-9.

- For the **Definitions File** prompt, the default is the special file name NUL.DEF, which tells LINK not to search for a definitions file.

File Naming Conventions

Use any combination of uppercase and lowercase letters for the file names you give in response to the prompts. For example, the following three file names are correct:

```
abcde.fgh
AbCdE.FgH
ABCDE.FGH
```

LINK uses the default file extensions .DEF, .DLL, .EXE, .LIB, .MAP, and .OBJ. You can cancel or replace the default extension by specifying a different extension.

To enter a file name that has no extension, type the name followed by a period. For example, if you type **ABC.** in response to a prompt, it tells LINK that the given file has no extension, but typing just **ABC** tells LINK to use the default extension for that prompt.

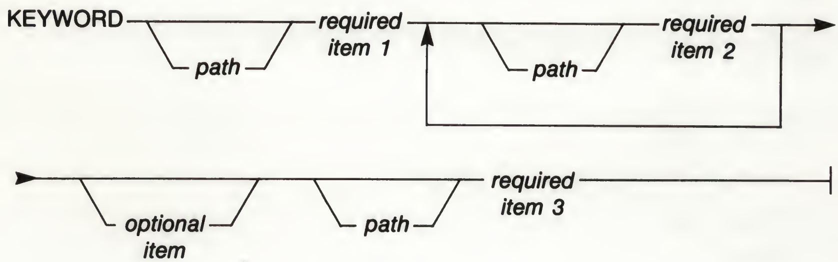
Type the name or names of the object files that you want to link. If you do not supply filename extensions, LINK uses .OBJ by default. If you have more than one name, separate each name with spaces or a plus sign (+). If you have more names than can fit on a single line, type a plus sign (+) as the last character on the line and press Enter. Link asks you for additional object files.

Using a Command Line to Specify LINK Files

Using Syntax Diagrams

These books use syntax diagrams to explain the format of commands entered on the operating system command line. The syntax diagram has the command name at the beginning (top left corner). You follow the diagram using the reading pattern of left-to-right and top-to-bottom.

The following is a sample syntax diagram:



Understanding Syntax Terms

syntax diagram	An illustration of possible structural patterns of a command sequence.
base line	A horizontal line that connects each of the required items in turn.
branch lines	Multiple horizontal lines that show choices. Branch lines are below the base line.
keyword	Words shown in all uppercase letters. The names of the compiler and other utilities are keywords. You can type keywords in any combination of uppercase and lowercase letters.
variable	Items shown in lowercase italic letters mean that you are to substitute the item. For example: *filename* indicates that you should type the name of your file in place of *filename*.
required items	Items that must be included. Required items appear on the base line. Command names are required items.
optional items	Items that you can include if you choose to do so. Optional items appear below the base line.
repeat symbol	A symbol that indicates you can specify more than one choice or a single choice more than once.

Arrows are used to show base line continuation and completion as follows:

The ——► symbol indicates that the command syntax is continued.

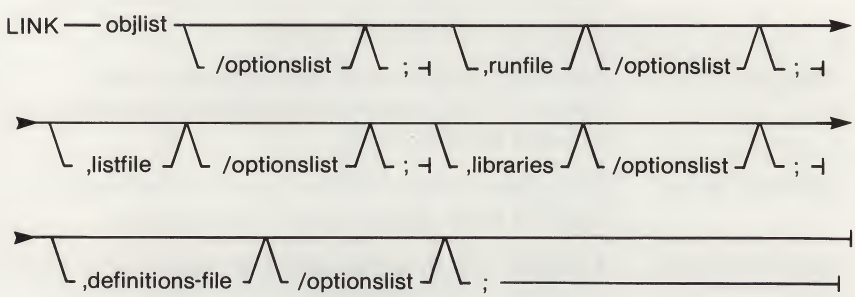
The ►— symbol indicates that a line is continued from the previous line.
The —┤ symbol indicates the end of a command.
The ↑_┘ symbol indicates that you can specify a choice more than once.

Reading a Syntax Diagram: To read a syntax diagram:

1. Start at the top left of the diagram.
2. Follow only one line at a time going from left to right and top to bottom.
3. Items on the lines indicate what you must or can specify and the required sequence.
4. When you encounter one or more branch lines, you must make a choice of items. Follow the line you choose from left to right except where you encounter the repeat symbol. The repeat symbol indicates you can make more than one choice or a single choice more than once.

With many commands, you can enter as many of a group of options as you want. These options are in a box that has a repeat arrow around it. You can follow the arrow and through the box until you have selected all the options you want to use. Once you have chosen an option from the box, you cannot choose the same option again.

You can create an executable program by typing LINK followed by the names of the files you wish to process, as shown:



To call the linker on the command line, give your responses to the command prompts by supplying the following information.

objlist	This is a list of object files that you want to link together, separated by plus signs or spaces. The linker requires at least one object file. If you do not supply an extension, the linker provides the extension .OBJ.
runfile	This is the name of the file to receive the executable output that the linker creates. If you do not supply a *runfile*, the linker creates a filename using the filename of the first object file in the command line and adding the extension .EXE for program modules and .DLL for dynamic link libraries. The filename can be prefixed with a drive and a path.
listfile	This is the name of the file to receive the link map. If you do not supply an extension, the linker provides the extension .MAP. If you specify the /MAP option, the linker creates a map file even if a map file is not specified in the command line. If the /MAP option and a map file are not specified, a map file is not created.
libraries	This is a list of libraries and directories for the linker to search, separated by plus signs or spaces.
definitions-file	This is an optional module definition file you can supply to the linker.
optionslist	If you specify options, you can put them anywhere on the command line. See "Using LINK Options" on page 4-20.

You can take the default for any parameter by omitting the filename or list. However, you must supply the comma (,) that would follow the filename or list. You can also select default responses by using the semicolon anywhere after the object list. The semicolon tells the linker to use the default responses for all remaining parameters.

If you do not specify a drive or directory for a file, the linker assumes the file is on the current drive and in the current directory. You must specify the location of each file.

If your application has more than one compiled source file, you must name all of them when you link. You can specify more than one object file. Separate multiple filenames by spaces or a plus sign (+).

If you do not supply all the filenames in the command line and do not end the line with a semicolon, the linker prompts you for additional files.

Examples Using a Command Line

This example uses an object module FILE.OBJ to create the executable file FILE.EXE. LINK searches the library ROUTINE.LIB for routines and variables used within the program. It also creates a file called FILE.MAP containing a list of the segments of the program and groups.

```
LINK
file.obj,file.exe,file.map,routine.lib;
```

It is equal to the following line:

```
LINK file , ,file, routine;
```

In the following example, the linker loads and links the object modules STATE.OBJ, CALIF.OBJ, TEX.OBJ, and MASS.OBJ and searches for unresolved references in the library file USALIB.LIB. By default, the executable file produced is named STATE.EXE. A list file named ROAD.MAP is also produced.

```
LINK STATE+CALIF+TEX+MASS,,ROAD,USALIB.LIB /MAP;
```

The next example uses the two object modules STARTUP.OBJ and FILE.OBJ on the current drive to create an executable file named FILE.EXE on drive B. The linker creates a map file in the /MAP directory of the current drive but does not search any libraries.

```
LINK startup+file,b:file,\map\file;
```

To link the application object file SAMPLE.OBJ using the module definition file SAMPLE.DEF and the libraries LIB1.LIB and LIB2.LIB, type:

```
LINK sample/A:4,sample.exe,sample.map/M,lib1+lib2/NOD,sample;
```

This command creates the file SAMPLE.EXE. It also creates the map file SAMPLE.MAP. The command searches the library files LIB1.LIB and LIB2.LIB to resolve any external references made in SAMPLE.OBJ. The /NOD option directs the linker to ignore any default libraries specified in the object file.

The linker uses default filename extensions if you do not explicitly provide them. In the example above, the linker extends the first occurrence of the file name SAMPLE to SAMPLE.OBJ and the final occurrence to SAMPLE.DEF. The linker extends the library files with the .LIB extension. /A:4 sets the segment-alignment factor to 4.

Using a Response File

A response file contains the names of all the files that you want processed. To operate the linker with a response file, you must create the file, then type the following:

LINK *@filename*

The *filename* is the name of the response file you created. It must be preceded by an **at** sign (@). If the file is in another directory or on another disk drive, you must provide a path name as well.

The response file has the following general form:

object modules
run file
list file
libraries
definitions file

Omit the elements that you have already provided at prompts or with a partial command line.

The responses must be in the same order as the linker prompts. Each response to a prompt must begin on a separate line, but you can extend long responses across multiple lines. To do so, type a plus sign (+) as the last character of each incomplete line. You can place options on any line. If you do not supply a filename for a group,

you must leave an empty line. Use options and command characters in the response file in the same way you type them on the keyboard.

You can place a semicolon on any line in the response file. When the linker reads the semicolon, it automatically supplies default filenames for all files not yet named in the response file. The linker then ignores the remainder of the response file (such as comments). For example, if you type a semicolon on the line of the response file corresponding to the **Run File** prompt, the linker uses the default responses for the executable file and for the remaining prompts.

When you use the LINK command with a response file, the linker displays each prompt on your screen with the corresponding response from your file. If the response file does not contain responses for all the prompts (in the form of filenames, the semicolon command character, or carriage returns), the linker displays the appropriate prompts and waits for you to enter the responses. When you type an acceptable response, the linker continues the link session.

Note: End a response file with either a semicolon (;) or a carriage-return/line-feed combination. If you do not provide a final carriage-return/line-feed in the file, the linker displays the last line of the response file and waits for you to press Enter.

Examples Using a Response File

The following lines in a response file tell the linker to load the four object modules STATE, CALIF, TEX , and MASS. The linker produces two output files, named STATE.EXE and ROAD.MAP. The /PAUSE option causes the linker to pause before producing the executable file (STATE.EXE). This permits you to change diskettes if necessary. See the /PAUSE and /MAP options under "The Map File" on page 4-83 for more information. The semicolon indicates there is no definition file.

```
STATE CALIF TEX MASS
/PAUSE /MAP
ROAD
USALIB.LIB;
```

The following response file tells the linker to link the four object modules MODA, MODB, MODC, and STARTUP. The linker pauses for you to change diskettes before producing the runfile MODA.EXE. The linker also creates a map file ABC.MAP and searches the library MATH.LIB in the \LIB directory of drive B:

```
moda modb modc startup
/PAUSE
abc
b:\lib\math;
```

Examples Using Mixed Methods

The following example combines all three methods of supplying file names. Assume that you have a response file called LIBRARY.ARF that contains the following line:

```
lib1+lib2+lib3+lib4;
```

Now start the linker with a partial command line:

```
LINK object1 object2
```

The linker takes OBJECT1.OBJ and OBJECT2.OBJ as its object files, and asks for the next line with the following:

```
Run File [object1.EXE]:  exec
List File [NUL.MAP]:
Libraries [.LIB]:  @library.arf
```

Type **exec** so that the linker names the run file EXEC.EXE and press the Enter key. Press Enter again to show that you do not want a map file. Type @ **library.arf** for the linker to use the response file containing the four library filenames.

The semicolon in LIBRARY.ARF indicates that there is no module-definition file for the link.

Using LINK Options

All linker options must begin with a forward slash (/). They can appear anywhere on the command or response line, as long as they come before the last comma on the line. If you are using a response file, you can place the options after the individual responses on the same line of the file or by themselves on a separate line.

When you specify more than one option, you can group them at the end of a single response to a prompt or distribute them among several responses for different prompts. Each option must begin with the forward slash character, even if other options precede it on the same line.

Linker options are named according to their function. You can abbreviate the names to save space. Make sure that your abbreviations are at least as long as the minimum abbreviations stated in "Linker Options" on page 4-21.

Abbreviations of option names must follow the same letter sequence as the original name. The linker allows no gaps or transpositions. For example, the following list of abbreviations would be valid for the /NOIGNORECASE option:

```
/NOI
/NOIG
```

Some linker options take numerical arguments. A numerical argument can be any of the following:

- A decimal number from 0 to 65535.
- An octal number from 0 to 0177777. The linker interprets a number as octal if it starts with a zero. For example, the number "10" is a decimal number, but the number "010" is an octal number, equal to 8 in decimal.
- A hexadecimal number from 0 to 0xFFFF. The linker interprets a number as hexadecimal if it starts with "0x". For example, "0x10" is a hexadecimal number, equivalent to 16 in decimal.

At the end of any command line parameter, you may specify one or more options to instruct the linker to perform a different function.

Linker Options

The following options can be used with the linker.

/ALIGNMENT

The ALIGNMENT option directs the linker to set the segment-alignment factor in the executable program to the number given. The number indicates a power of 2. The default value is 9 (512 bytes).

Each file segment in the executable file will start on a file offset which is a multiple of the amount specified in the /ALIGNMENT option.

Note that this option does not affect the align type of a segment when combining that segment with other segments into a single segment.

The smaller the alignment number, the smaller the executable module size. The larger the alignment number, the greater the padding. The smallest number possible should be used.

This option is valid for linking OS/2 mode executable files only.

Format

/ALIGNMENT:*number*

The minimum abbreviation is **/A**.

Remarks

The *number* can be a hexadecimal, decimal, or octal number. For IBM FORTRAN/2, a value of 8 is recommended but not required.

This option allows you to change the default value of the MAXALLOC field, which controls the maximum number of paragraphs reserved in storage for your program. A paragraph is defined as the smallest storage unit (16 bytes) addressable by a segment register.

Format

/CPARMAXALLOC:*number*

The minimum abbreviation is **/CP**.

Remarks

The maximum number of paragraphs reserved for a program is determined by the value of the MAXALLOC field at offset 0xC in the EXE header. For more information about .EXE file structuring and loading, see the IBM Personal Computer *DOS Technical Reference* manual.

By default, the MAXALLOC field is set to 65535 (decimal). However, you can set the value to any number between 1 and 65535 (decimal, octal, or hexadecimal). This is useful when you know that the efficiency of your program is not increased by reserving all available storage, or when you want to run another program from within your program and additional storage is necessary.

If the value you specify is less than the computed value of MINALLOC (at offset 0xA), the linker uses the value of MINALLOC instead.

This option has no effect on FORTRAN/2 object, since any reserved storage will be returned to DOS or OS/2 when the program is executed.

/DOSSEG

The /DOSSEG option forces segments to be ordered according to the following rules:

1. All segments with a class name ending in CODE
2. All other segments outside of DGROUP
3. DGROUP segments in the following order:
   a. Any segments of class BEGDATA (this class name is reserved for IBM use)
   b. Any segments not of class BEGDATA, BSS, or STACK
   c. Segments of class BSS
   d. Segments of class STACK

This option is not needed with IBM FORTRAN/2 but may be specified when linking with objects of other languages when appropriate.

Format

/DOSSEG

The minimum abbreviation is /DO.

This option directs the linker to remove sequences of repeated bytes (typically nulls) and to optimize the load-time relocation table before creating the DOS executable file.

This option is valid for linking DOS mode .EXE files only.

Format

/EXEPACK

The minimum abbreviation is /E.

Remarks

Executable files linked with this option are usually smaller and load faster than files linked without it. You may use the FORTRAN/2 Interactive Symbolic Debug with packed files. However, you might not be able to use symbolic debugger programs that are available for use with other languages with packed files.

The /EXEPACK option does not always save a significant amount of disk space and may sometimes actually increase file size. Programs that have a large number of load time relocations (about 500 or more) or long streams of repeated characters are usually made shorter when they are packed.

This option is recommended for IBM FORTRAN/2 unless the size of the executable is increased (which is not usually the case).

Example

This example creates a packed version of file PROGRAM.EXE.

```
LINK  program  /E;
```

/FARCALLTRANSLATION

The /FARCALLTRANSLATION option directs LINK to optimize intra-segment far calls into the sequence:

```
NOP
PUSH CS
CALL NEAR address
```

By default, the linker does not perform this optimization.

In most programs, this option will yield significant savings in executable size and load time. However, there is a small chance that, during optimization, the linker will mistakenly identify a byte with a value of `0x9a` as a far call, when in fact, it is an assembled constant. You should use this option with caution.

This option is valid for linking OS/2 mode executable files only.

Format

/FARCALLTRANSLATION

The minimum abbreviation is **/FAR**.

The /HELP option causes the linker to write a list of the available options to the screen. This may be convenient if you need a reminder of the available options. When you use this option, do not give any filenames in the LINK command.

Format

/HELP

The minimum abbreviation is **/HE**.

Example

```
LINK /HELP
```

/INFORMATION

The /INFORMATION option causes the linker to display which phase of processing it is running, and the name of each input module as it is linked. This is useful during debugging.

Format

/INFORMATION

The minimum abbreviation is /I.

The /MAP option causes the linker to produce a listing of all public symbols declared in your program. This is in addition to the map of program segments generated when you specify a map file in the LINK command. The listing is added to the map file created by the linker.

Format

/MAP [:*number*]

The minimum abbreviation is **/M**.

For a complete description of the listing-file format, see "The Map File" on page 4-83 in this chapter.

Remarks

If you do not specify a map file in the LINK command, you can use the /MAP option to force the linker to create a map file. The linker gives the forced map file the same name as the first object file specified in the command and the default extension .MAP.

The *number* parameter specifies the maximum number of public symbols that the linker can sort in the map file. If you give no number, the limit is 2048.

If you get the following linker error message:

Too many public symbols

You must link again with a higher number.

If the number parameter is greater than 32KB, or there is not enough memory to increase the limit to the requested value, the linker issues the following error and produces an unsorted list.

MAP symbol limit too high

If you get this error, link again with a lower number. The limit varies according to how many segments the program has and how much memory is available.

Specifying a number has an additional effect. The public symbols are sorted by address only and not by name, regardless of the number specified. If you want to reduce the size of your map files by removing the list sorted by name, link with /MAP followed by a number large enough to accommodate the number of public symbols in your program.

If you do not specify a map file in a LINK command, you can use the /MAP option to force the linker to create a map file. LINK gives the forced map file the same name as the first object file specified in the command and the default extension .MAP.

See "The Map File" on page 4-83 for examples of map files.

/NODEFAULTLIBRARYSEARCH

The /NODEFAULTLIBRARYSEARCH option directs the linker to ignore any library names it may find in an object file. IBM FORTRAN/2 adds a library name to an object file to ensure that a default library is linked with the program. Using this option bypasses this default library and lets you name the libraries you want by including them on the linker command line.

Format

/NODEFAULTLIBRARYSEARCH

The minimum abbreviation is **/NOD**.

Example

This example links the object files WAKEUP.OBJ and TIMER.OBJ with routines from the libraries RUNTIME and DEBUG. Any default libraries that may have been named in WAKEUP.OBJ or TIMER.OBJ are ignored.

```
LINK WAKEUP+TIMER/NOD,,,DEBUG+RUNTIME;
```

Note: This option is not recommended for use with IBM FORTRAN/2 except for advanced applications.

/NOFARCALLTRANSLATION

The /NOFARCALLTRANSLATION option directs the linker to disable translation of intra-segment far calls.

This option is in effect by default.

This option is valid for linking OS/2 mode executable files only.

Format

/NOFARCALLTRANSLATION

The minimum abbreviation is **/NOF**.

See "/FARCALLTRANSLATION" on page 4-26 for more information.

The /NOIGNORECASE option directs the linker to treat uppercase and lowercase letters in symbol names as distinct letters. Normally, the linker considers uppercase and lowercase letters to be identical, treating the names *TWO*, *Two*, and *two* as the same name. When you use the /NOIGNORECASE option, the linker treats *TWO*, *Two*, and *two* as different names.

Format

/NOIGNORECASE

The minimum abbreviation is **/NOI**.

Remarks

IBM FORTRAN/2 converts all names to uppercase and all names in its libraries are in uppercase. For IBM FORTRAN/2 to call routines written in other languages which have lowercase names, the option should *not* be specified. If this option is specified, IBM FORTRAN/2 will be unable to call any routines which have lowercase names. Routines written in other languages that are case specific should have uppercase entry names if they are to be called by IBM FORTRAN/2.

Example

```
LINK file+file2/NOI;
```

This command causes the linker to treat uppercase and lowercase letters in symbol names as distinct letters. The object files FILE.OBJ and FILE2.OBJ are linked with routines from the FORTRAN language library.

/NOPACKCODE

This option directs the linker to disable the packing of code segments. /NOPACKCODE is the opposite of /PACKCODE. By default, the linker packs code segments. See "/PACKCODE" on page 4-35 for more information.

This option is valid for linking OS/2 mode executable files only.

Format

/NOPACKCODE

The minimum abbreviation is **/NOP**.

This option directs the linker to try to pack adjacent logical code segments into one physical segment. By default, the linker packs code segments. To suppress packing, see "/NOPACKCODE" on page 4-34.

This option is valid for linking OS/2 mode executable files only.

Format

/PACKCODE[:*number*]

The minimum abbreviation is **/PAC**.

Remarks

The optional *number* is the limit at which to stop packing. If no number is given, the linker uses 65536. For more information on packing, see "Rules for Segment Packing in LINK" on page 4-47.

This option is for performance. Smaller segments allow only the code that is being used to be in memory. However, this requires more overhead to maintain. Larger segments cost less to maintain but may put more code in memory than is actually needed.

When you pack segments you may cut down on the amount of padding between segments (needed by file segment alignment, see "/ALIGNMENT" on page 4-22), and relocation records. This saves space and load time. Also, you reduce the number of inter-segment calls during execution of the program, which decreases the execution time. In the typical FORTRAN program, any advantages of keeping only the minimum amount of code in memory (that is, by not packing) are probably outweighed by the advantage of packing. See also "Rules for Segment Packing in LINK" on page 4-47.

/PAUSE

The PAUSE option causes the linker to pause before writing the executable file to disk. This enables you to change diskettes if necessary.

Format

/PAUSE

The minimum abbreviation is **/PAU**.

Remarks

If you choose the /PAUSE option, the linker displays the following message before creating the run file:

About to generate .EXE file
Change diskette in drive *n* and press Enter

Note that *n* is the appropriate drive letter. This message appears after the linker has read the data from the object files and library files and has written it to the map file. The linker resumes processing when you press Enter. After the linker writes the executable file to disk, the following message appears:

Please replace original diskette
in drive *n* and press Enter

Note: Do not remove the disk used for the temporary disk file, if one has been created. If the temporary disk message appears when you have specified the /PAUSE option, you should press CTRL-BREAK to end the linker session. Rearrange your files so that the linker can write the temporary file and the executable file to the same disk, then repeat the procedure. See "The Temporary Disk File" on page 4-52 for more information about the temporary disk file.

Example

This command causes the linker to pause just before creating the executable file FILE.EXE. After creating the executable file, the linker pauses again to let you replace the original disk.

```
LINK file/PAUSE,file,,\lib\math;
```

/SEGMENTS

The /SEGMENTS option directs the linker to process no more than a specified *number* of logical segments per program. If it meets more than the given limit, the linker displays an error message and stops linking. The /SEGMENTS option bypasses the default limit of 128 logical segments.

Format

/SEGMENTS:*number*

The minimum abbreviation is **/SE**.

Remarks

If you do not specify /SEGMENTS, the linker reserves enough storage space to process up to 128 segments. If you get the following linker error message:

Too many segments

You must set the segment limit higher to increase the number of segments the linker can process.

If you get the following linker error message:

Segment limit set too high

Set the segment limit lower.

If you cannot increase the limit and you still have too many segments, you must reduce the number of segments in the program by combining segments, that is, reduce the number of program units and common blocks. You may also try increasing the amount of memory in your system if you do not already have the maximum allowed.

The *number* can be any integer value in the range 1 to 3072. It must be a decimal, octal, or hexadecimal number. Octal numbers must have a leading zero. Hexadecimal numbers must start with a leading zero followed by a lowercase *x*. For example, 0x4B.

In general, each program unit requires two segments: a code segment and a data segment. In addition, each common block in the program requires a segment.

Local data and common may require additional segments if they are larger than 64KB.

See "Compiler Generated and Library Segment Names" on page 8-18 for more details on segments.

Examples

This example sets the segment limit to 192:

```
LINK file/SE:192;
```

The next example sets the segment limit to 255 using a hexadecimal number:

```
LINK
fortmain+fortsub,run/SEGMENTS:0xff,mainsub,em+mlibfp;
```

/STACK

The /STACK option sets the program stack to the number of bytes given by *size*. Usually, the linker calculates the stack size of a program, as the sum of the size of any stack segments given in the object files. The stack segment is the first segment seen by the linker which has combine type stack. If you specify /STACK, the linker uses the given *size* in place of the value it calculates.

Format

/STACK:*size*

The minimum abbreviation is **/ST**.

Remarks

The *size* can be any positive integer value in the range 1 to 65534. The value can be a decimal, octal, or hexadecimal number. Octal numbers must begin with a zero. Hexadecimal numbers must begin with a leading zero followed by a lowercase *x*. For example, 0x1B.

If size is not a multiple of two, size plus one is used for the size of the program stack.

If both the /STACK option and the STACKSIZE module-definition statement are given, STACKSIZE overrides the /STACK option.

If no stack segment is defined and no STACK or STACKSIZE is given, then DOS mode .EXE files do not get a stack and OS/2 mode .EXE files get their initial SS:SP field set to DS:0 where DS is the automatic data segment (DGROUP). When loaded, the OS/2 program loader sets SP to the end of the automatic data segment. (An OS/2 mode .EXE file must have an automatic data group, otherwise, an error occurs.)

See also "/STACK" on page 4-40, "STACK Segment" on page 8-5 and, the module definition statement "STACKSIZE" on page 4-81.

Examples

The first example sets the stack size to 512 bytes.

```
LINK file/STACK:512;
```

The second example sets the stack size to 255 bytes using a hexadecimal number.

```
LINK moda+modb,run/ST:0xFF,ab\lib\start;
```

The final example sets the stack size to 24 bytes, using an octal number.

```
LINK startup+file/ST:030;
```

/WARNFIXUP

The /WARNFIXUP option directs the linker to issue a warning for each segment-relative fixup of location-type ''offset'', such that the segment is contained within a group but is not at he beginning of the group. The linker will include the displacement of the segment from the group in determining the initial value of the fixup, contrary to what happens with DOS mode .EXE files.

This option is valid for linking OS/2 mode executable files only.

Format

/WARNFIXUP

The minimum abbreviation is **/w**.

Advanced Linker Topics

How the Linker Works

The linker performs the following steps to combine object modules and produce a run file:

1. Reads the object modules you submit
2. Searches the given libraries, if necessary, to resolve external references
3. Assigns addresses to the segments
4. Assigns addresses to the public symbols
5. Reads data in the segments
6. Reads all relocation references in the object modules
7. Performs fix-ups
8. Outputs a run file (executable image) and relocation information.

The linker produces a list file that shows segment and public symbol addresses and most error messages.

The executable image contains the code and data that make up the executable file.The relocation information is a list of references to locations in the program whose final address is not determined until the program is loaded by the operating system. The format and processing of those relocations is different for OS/2 and DOS.

Order of Segments

Normally, the linker copies segments to the executable file or dynamic link library in the same order that it meets them in the object files. This order is maintained throughout the program unless the linker finds two or more segments having the same class name. Segments having identical class names belong to the same class type, and are copied (or combined) to the executable files as contiguous segments. See "Compiler Generated and Library Segment Names" on page 8-18 for the segment names, class names, combine types, align types, and groups generated by IBM FORTRAN/2.

The order described above can be altered by using the "/DOSSEG" on page 4-24, or by the module definition statement "SEGMENTS" on page 4-79.

Combined Segments

The linker uses combine types to tell if two or more segments that are sharing the same segment name should be treated as one physical segment. The combine types are **public**, **stack**, **common**, and **private**.

If a segment is combine type **public**, the linker combines it with any segments of the same name and class. When the linker combines segments, it makes the segments contiguous in storage; therefore, you can reach each address in the segments using an offset from one the beginning of the combined segment. The result is the same as if the segments were defined as a whole in the source file.

The linker preserves the align type of each segment in the combined segment. So, even though the individual segments compose a single, larger segment, the code and data in each segment retain the original align type of the segment. If the linker tries to combine segments that total more than 64KB, it displays an error message.

If a segment is combine type **stack**, the linker combines individual segments as it does for **public** combine types. However, for **stack** segments, the linker copies an initial stack-pointer value to the executable file. This stack-pointer value is the offset to the end of the first **stack** segment (or combined **stack** segment) that the linker meets. If you use the **stack** type for **stack** segments, you need not give instructions to load the segment selector into the **SS** register.

If a segment is combine type **common**, the linker combines it with any segments of the same name and class. When the linker combines **common** segments, it places the start of each segment at the same address. This creates a series of overlapping segments. The resulting combination segment has a length equal to the length of the longest individual segment.

The linker assigns a default combine type **private** to any segment without an explicit combine type definition in the source file. The linker does not combine **private** segments.

See also "/PACKCODE" on page 4-35 and "/NOPACKCODE" on page 4-34 to combine code segments having different segment names into one physical segment.

Groups

For DOS mode executables, **groups** let segments of various classes be addressable relative to the same frame address. When the linker encounters a **group**, it adjusts all storage references to items in the **group** so that they are relative to the same frame address.

Segments of a group need not be contiguous, belong to one class, or have the same combine type. All segments of the **group** must fit within 64KB of storage.

For OS/2 mode executables, a group is synonymous with a selector or physical segment. The segments with a group must be contiguous.

Groups do not affect the order of loading of segments. You must use class names and enter object files in the correct order to guarantee contiguous segments. The /dosseg option or the segments module-definition statement may also be used to control the order. If the **group** is smaller than 64KB of storage, the linker may place segments that are not part of the **group** in the same storage area. For DOS mode executable files, the linker does not specifically check that all segments in a group fit within 64KB of storage; if the segments are larger than the 64KB maximum, the linker can produce a **fixup-overflow** error.

A description of **groups** and defining **groups** is in the IBM *Macro Assembler/2 Language Reference* manual.

Fixups

Once the linker knows the starting address of each segment in a program and establishes all segment combinations and groups, it can resolve ("fix up") any unresolved references to labels and variables. The linker computes an appropriate offset and segment address and replaces the temporary address values with the new values.

The size of the value that the linker computes depends on the type of reference. If the linker discovers an error in the anticipated size of

the reference, it displays a fixup-overflow error message. This happens, for example, when a program tries to use a 16-bit offset to address an instruction in a segment that has a different frame address. It also occurs when the segments in a group do not fit within a single, 64KB block of storage.

The linker resolves the following four types of references:

- Short
- Near self-relative
- Near segment-relative
- Long.

In the following for DOS mode executable files, a *frame* is a contiguous region of 64KB of memory, beginning a paragraph boundary (that is, on a multiple of 16 bytes). Every frame begins a *paragraph* boundary. The paragraphs in memory are numbered from 0 to 65535. These numbers, each of which define a frame, are called *frame numbers*. Any location in memory is contained in exactly 4096 distinct frames; but one of these frames can be distinguished because it has a higher frame number. This distinguished frame is called the canonical frame of that location.

By extension, if s is any set of memory locations, then there exists a unique frame which has the lowest frame name is the set of canonical frames of the location. This unique frame is called *canonical frame* of the set s. Thus we may speak of the canonical frame of a logical segment or of a group of logical segments.

A **short** reference occurs in JMP instructions that try to pass control to labeled instructions that are in the same segment or group. The target instruction must not be longer than 128 bytes from the point of reference. The linker computes a signed, 8-bit number for this **short** reference. It displays an error message if the target instruction belongs to a different segment or group (different frame address). The linker also displays an error message if the distance from the frame address to the target is more than 128 bytes in either direction.

A **near self-relative** reference occurs in instructions that access data relative to the same segment or group. The linker computes a 16-bit offset for this **near self-relative** reference. It displays an error message if the data resides in more than one segment or group.

A **near segment-relative** reference occurs in instructions that attempt to access data either in a specified segment or group, or relative to a specified segment register. The linker computes a 16-bit offset for this **near segment-relative** reference. It displays an error message if the offset of the target within the specified frame is greater than 64KB or less than 0 bytes. The linker also displays an error message if the beginning of the canonical frame of the target is not addressable.

A **long** reference occurs in CALL instructions that try to get access to an instruction in another segment or group. The linker computes a 16-bit frame address and a 16-bit offset for this **long** reference. The linker displays an error message if the computed offset is greater than 64KB or less than 0 bytes. The linker also displays an error message if it cannot address the beginning of the canonical frame of the target.

Fixups for OS/2 mode executables and dynamic libraries are the same as in DOS mode.

Rules for Segment Packing in LINK

When producing an OS/2 mode executable file, the linker may pack distinct adjacent segments into the same physical segment. The physical segment is represented by an entry in the program segment table. The rules that the linker uses when packing segments are as follows:

- The limit of the total size of a set of segments packed into a physical segment is 64KB. When the size of a physical segment reaches 64KB, the linker starts a new physical segment.
- The linker packs adjacent segments only into a physical segment.
- The linker packs segments in the same group into a physical segment. The linker does not pack segments in different groups into a physical segment. An error results if a segment in one group occurs between segments in another group.
- If you use the /PACKCODE:*packlimit* option, (which is the default), the linker packs code segments to the size you specify in the *packlimit*. The default *packlimit* is 64KB. If you use the /NOPACKCODE option, the linker will not pack code segments unless they are in the same group.

- If the segment type (CODE or DATA) is not the same in all the packed segments, the linker sets the physical segment flags to NONSHARED CODE. The segment type is determined from the default segment attributes specified in the module-definitions file. The default segment attribute is CODE if the class name ends with 'code' (ignoring case) and otherwise the attribute is DATA.
- If the linker packs any segments that have an attribute of NONSHARED, it sets the physical segment flags NONSHARED CODE.
- If the linker packs any segments that are NOT ERONLY (execute/read only), it marks the entire physical segment as NOT ERONLY.
- If the linker packs any segments that are PRELOAD, it designates the entire physical segment as PRELOAD.

Searching Directories for Libraries

You can direct the linker to search directories and disk drives for the libraries you have named in a command or response. This is done by specifying one or more search paths with the library names, or by assigning the search paths to the environment variable LIB before you call the linker. Environment variables are explained under the SET command in the IBM Personal Computer *DOS Reference* manual.

A *search path* is the path specification of a directory or drive name. You enter search paths along with library names on the link command line or in response to the **Libraries** prompt. You can specify up to 16 search paths. You can also assign the search paths to the LIB environment variable using the DOS SET command. In the latter case, you must separate the search paths with semicolons.

If you include a driver or directory name in the file name for a library in the link command line, the linker searches that drive or directory only. If you do not specify a drive or directory, the linker searches for library files in the following order:

1. First, the linker searches the current drive and directory.
2. If the linker cannot find the library and you have given one or more search paths in the command line, the linker searches along the specified search paths in the order in which you specified them.

3. If the linker still cannot find the library and you set a search path with the LIB environment variable, the linker searches there.
4. If the linker still cannot find the library, it prints an error message.

Examples

In the first example, the linker searches only the \ALTLIB directory on drive A to find the library MATH.LIB. To find COMMON.LIB, the linker must search the current directory on the current drive, the current directory on drive B, and finally, directory \LIB on drive D.

```
LINK file,,file,A:\altlib\math.lib+common+B:+D:\lib\;
```

In the second example, the linker searches the current directory, directory \LIB on drive C, and directory \SYSTEM\LIB on drive U to find the libraries MATH.LIB and COMMON.LIB.

```
SET LIB=C:\lib;U:\system\lib
LINK file,,file.map,math+common;
```

Default Library and the Library Search Path

The IBM FORTRAN/2 compiler encodes object files that it creates with the name of a default library. This encoded information enables the linker to search for the default library files and link them with your program.

You do not have to give the name of the default library file when you link. However, you must specify the directory or directories where the library file resides, unless they are in the current directory. You can do this by giving directory specifications following the linker **Libraries** prompt, by setting the LIB environment variable, or by a combination of the directory specification and the LIB environment variable.

Following the the linker **Libraries** prompt, you can specify more than one directory, or you can choose the default by not specifying any directory.

Note: Each directory specification must end with a backslash (\) or colon so that the linker can recognize the specification as a directory name or drive letter rather than a library name.

The LIB variable can contain one or more directory specifications.

To locate library files, the linker:

1. Searches the current working directory.
2. Searches any directories specified in the command or after the **Libraries** prompt, if the library files are not found. The linker searches the directories in order of their appearance on the line.
3. Searches the libraries specified by the LIB environment variable if it does not find the library files. The linker searches the directories in order until the given libraries are found.

You can separate library files and store them in different directories because the linker searches as many of the specified directories as necessary to find the files.

If you want to link additional libraries, specify the library names following the **Libraries** prompt or in the command. The linker uses the same procedure to search for additional libraries as it does for the default libraries. However, if you give a library name that includes a pathname, the linker searches just that path name for the library; no other directory specifications apply.

Moving or Discarding Code Segments Under OS/2

OS/2 will take best advantage of storage by moving and discarding code segments, and moving and swapping data segments.

Notes:

1. If the application must use a local heap, you can reserve heap space at run time by using the OS/2 service DOSREALLOCSEG to increase the size of the automatic data segment. The *automatic data segment* is the group of logical segments, defined with the IBM Macro Assembler/2 pseudo-op GROUP, named DGROUP.
2. IBM FORTRAN/2 does not use heap space.

Using Overlays

In DOS mode .EXE files, you can direct LINK to create an *overlaid* version of your program. This means that LINK loads parts of your program only when they are needed. The overlaid version shares the same space in storage. Your program should be able to run in less storage, but usually runs more slowly because of the time needed to read and load the code into storage.

Note: You cannot create an overlaid version of your program from object modules created with IBM FORTRAN/2. Overlays are supported when linking object modules created with other IBM high level languages.

You specify the overlay structure to the linker in response to the **Object Modules** prompt. Loading is automatic. You specify the overlays in the list of modules that you submit to the linker by enclosing them in parentheses. Each parenthetical list represents one overlay. For example:

```
Object Modules [.OBJ]: a+(b+c)+(e+f)+g+(i)
```

The elements (b + c), (e + f), and (i) are overlays. The remaining modules, and any drawn from the run time libraries, make up the resident or root part of your program. LINK loads your program or root overlays into the same region of storage, so only one can be resident at a time. Because LINK does not allow duplicate names in different overlays, each module can occur only once in a program.

The linker replaces calls from the root to an overlay and calls from an overlay to another overlay with an interrupt, followed by the module identifier and offset. The DOS interrupt number is 3FH.

Restrictions

LINK adds the name for the overlays to the .EXE file, and encodes the name of this file into the program so the overlay manager can get access to it. If, when the program is initiated, the overlay manager cannot find the .EXE file (perhaps it was renamed or is not in a directory specified by the PATH environment variable), the overlay manager prompts you for a new name.

You can only overlay modules to which control is transferred and returned by a standard 8088 long (32-bit) CALL/RETURN instruction.

You cannot use long jumps or indirect calls (through a function pointer) to pass control to an overlay. When a pointer calls a function, the called function must either be in the same overlay or in the root.

Do *not* place object modules that have been compiled with the /T option in overlays. Only code is overlayed. Data is not overlayed.

Overlay Manager Prompts

In the following example, suppose that B is the default drive:

```
Cannot find PAYROLL.EXE

Please enter new program spec:
```

The response `EMPLOYEE\DATA\` causes the overlay manager to look for `EMPLOYEE\DATA\PAYROLL.EXE` on drive B.

Suppose that you must change the diskette in drive B. If the overlay manager needs to swap overlays, it finds that PAYROLL.EXE is no longer on the B drive and gives the following message:

```
Please put diskette containing
B:\EMPLOYEE\DATA\PAYROLL.EXE in drive
B: and strike any key when ready.
```

After the overlay is read from the diskette, the overlay manager gives the following message:

```
Please restore the original diskette.  Strike any key when ready.
```

The Temporary Disk File

The linker uses available storage for the link session. If the files to be linked create an output file that exceeds available storage, the linker creates a temporary disk file (named VM.TMP) to serve as storage. The linker also creates the temporary file in the current working directory and displays the following message:

Temporary file *name* has been created.
Do not change diskette in drive *letter*:

The *name* is a unique temporary filename created by the linker. After this message appears, do not remove the diskette from the given drive (*letter*) until the link session ends. If you remove the diskette, the operation of the linker is unpredictable, and you may see the following message:

Unexpected end of file on *name*

If you get this message, you must restart the link session from the beginning.

After the linker creates the executable file, it automatically deletes the temporary file.

Note: Do not use the file name VM.TMP for your own files. When the linker creates the temporary file, it destroys any previous file having the same name.

Linking for IBM FORTRAN/2

Libraries and dynamic link libraries are included with IBM FORTRAN/2 to support execution on OS/2 and DOS. Object files generated by the compiler library include a search record to the appropriate default library. The appropriate default libraries should be installed on the system where the link is done.

On OS/2, the library DOSCALLS.LIB must be specified for the link if Application Programming Interface functions are called from a program.

The object files are independent of the target or current operating system version. Linking requires specifying the directories containing the appropriate libraries.

The creation of Family Application Programming Interface programs requires the BIND utility and the API.LIB library from the OS/2 Toolkit (not provided with FORTRAN/2). The creation of Family Application Programming Interface programs is only supported on OS/2.

.EXE files to execute under OS/2 provide support for true segmented programs and protected address mode. These .EXE files provide information per segment and per entry point to support dynamic linking.

Object Libraries and Dynamic Link Libraries

The following libraries for execution support with IBM Math Co-Processor emulation are supplied on the master diskettes:

Name	Contains
FORTRIE.LIB	Linkable object used at intrinsic functions
FORTRRE.LIB	Linkable objects used at runtime, debug, and emulator
FORTRDE.LIB	Dynamic links to runtime, debug, and emulator

The following libraries for execution support with the math coprocessor are supplied on the master diskettes:

Name	Purpose
FORTRIN.LIB	Linkable objects used at intrinsic functions
FORTRRN.LIB	Linkable objects used at runtime and debug
FORTRDN.LIB	Dynamic links to runtime and debug

The following library for execution support on DOS is supplied on the master diskettes:

PCDOS.LIB

This library contains the Family Application Programming Interface mappings needed for intrinsic functions, runtime, and debug. The mappings are not supported for direct use by applications.

The following libraries are built during installation for making DOS mode .EXE files:

Name	Contents
FORTRAE.LIB	FORTRIE.LIB + FORTRRE.LIB + PCDOS.LIB
FORTRAN.LIB	FORTRIN.LIB + FORTRRN.LIB + PCDOS.LIB

The following libraries are built during installation for making OS/2 mode .EXE files:

Name	Contents
FORTRAE.LIB	FORTRIE.LIB + FORTRDE.LIB
FORTRAN.LIB	FORTRIN.LIB + FORTRDN.LIB

Note: These two libraries have the same names as the DOS mode libraries above so that the default link on each system will find the correct libraries. The default installation is to place the OS/2 mode libraries in a different directory or on a different diskette.

The following libraries are built during installation for making Family Application Programming Interface (OS/2 mode/DOS mode) .EXE files:

Name	Contents
FORTRAE.LIB	FORTRIE.LIB + FORTRRE.LIB
FORTRAN.LIB	FORTRIN.LIB + FORTRRN.LIB

Note: These two libraries have the same names as the DOS mode and OS/2 mode libraries above so that the default link will find the appropriate libraries. The default installation is to place the Family Application Programming Interface libraries in a different directory or on a different diskette.

The following dynamic link libraries are supplied on the master diskettes to support the dynamic links when running in OS/2 mode:

Name	Purpose
FORTRUE.DLL	Runtime, debug, and emulator for math coprocessor emulation (objects from FORTRRE.LIB)
FORTRUN.DLL	Runtime and debug for math coprocessor (objects from FORTRRN.LIB)

The appropriate .DLL file must be available when a program with dynamic links is executed in OS/2 mode. The file will be searched for in the directories specified by the LIBPATH command in the CONFIG.SYS file.

Linking OS/2 Mode and DOS Mode Applications

The linker can link code for running in DOS or OS/2 mode or real (compatibility) mode. The linker determines which form of executable file to produce by the presence of dynamic link entries and definition files. If a dynamic link entry resolves any reference or if you request a definition file, then the linker produces an OS/2 executable file; otherwise, it produces a DOS executable file.

The IBM FORTRAN/2 compiler encodes the name of a default library in the object file it creates. The default library selected depends on whether or not the /N option is specified.

If the OS/2 mode libraries are used, a OS/2 mode .EXE file will be generated.

If the DOS mode libraries are used, a DOS mode .EXE file will be generated.

Linking can be done in either OS/2 mode or DOS mode independently from the desired type of .EXE.

If the /N option is selected, the following default library will be selected:

FORTRAE.LIB

This library uses the emulator for math coprocessor emulation.

If the /N option is not selected, the following default library will be selected:

FORTRAN.LIB

This library uses the math coprocessor.

You do not have to give the names of the default library file when you link. However, you must specify the directory where the library resides. For OS/2 mode .EXE files, the library DOSCALLS.LIB and the directory in which it resides must also be specified if Application Programming Interface functions are called from the program. (DOSCALLS.LIB is supplied with OS/2.) You can do this by giving directory and library specifications in the LINK command following its LINK **Libraries** prompt, by setting the LIB environment variable, or by a combination of the **Libraries** prompt and the LIB environment variable.

Following the LINK **Libraries** prompt, you can specify more than one directory or library, or you can take the default by specifying nothing (for an OS/2 mode .EXE file, however, you may need to specify the library DOSCALLS.LIB).

If you explicitly specify the wrong libraries, the linker may flag the following names:

```
F@EMUL or F@8087
```

as undefined. F@EMUL being undefined indicates that the library for non-emulation was found for object files with floating point instructions that were compiled with the /N option. F@8087 being undefined indicates that the library for emulation was found for object files with floating point instructions that were compiled without the /N option.

Linking without errors does not necessarily indicate the correct libraries were used.

Creating Family Applications

You can produce executable files that can run in either real or OS/2 mode (a Family Application). To do this, use the BIND utility and an application programming interface mapping library. BIND is not supplied with IBM FORTRAN/2. (It is part of the OS/2 Toolkit.)

Creation of Family Application Programming Interface programs is only supported on OS/2.

With dynamic runtime, the program should be linked using the directory containing default libraries for OS/2 mode and specifying the library DOSCALLS.LIB as needed.

The resulting .EXE file should then be processed by BIND. This requires access to APLI.LIB, DOSCALLS.LIB, and for a program compiled with the /N option:

```
FORTRAE.LIB
```

or for a program compiled without the /N option:

```
FORTRAN.LIB
```

The above libraries should be taken from the Family Application Programming Interface directory or diskette. BIND and API.LIB are from the OS/2 Toolkit. (This is not provided as part of IBM FORTRAN/2).

If the wrong libraries are specified, the linker may flag the following names:

F@EMUL or F@8087

as undefined. F@EMUL being undefined indicates that the library for non-emulation was found for object files with floating point instructions that were compiled with the /N option. F@8087 being undefined indicates that the library for emulation was found for object files with floating point instructions that were compiled without the /N option.

Without dynamic runtime, the program should be linked using the default libraries in the Family Application Programming Interface directory or diskette and explicitly specifying the DOSCALLS.LIB library.

If you explicitly specify the wrong libraries, the linker may flag the following names:

F@EMUL or F@8087

as undefined. F@EMUL being undefined indicates that the library for non-emulation was found for object files with floating point instructions that were compiled with the /N option. F@8087 being undefined indicates that the library for emulation was found for object files with floating point instructions that were compiled without the /N option.

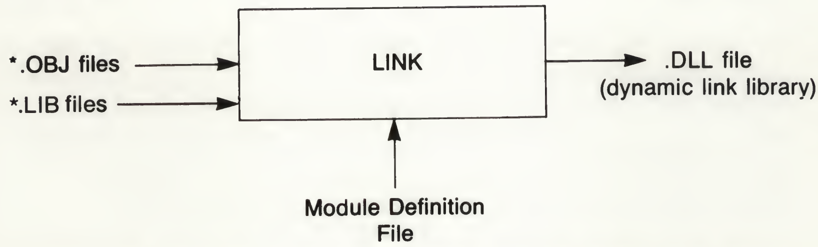
Linking without errors does not necessarily indicate the correct libraries were used.

The resulting .EXE file should then be processed by BIND. This requires access to APLI.LIB and DOSCALLS.LIB.

BIND and API.LIB are from the OS/2 Toolkit. (This is not provided as part of IBM FORTRAN/2.)

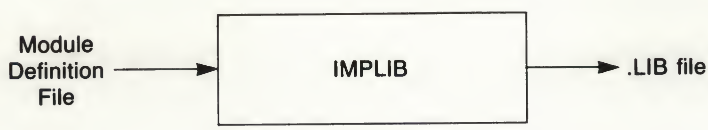
Creating Dynamic Link Libraries

You create OS/2 dynamic link libraries by linking your compiled source files (.OBJ files), the appropriate libraries (.LIB files) and a module-definition file that contains a LIBRARY statement using the LINK utility. The resulting dynamic link library contains entry points, which you can call under from an OS/2 mode application or from other dynamic link libraries.



Use the IMPLIB utility to create a .LIB file for the dynamic link library. You can use the .LIB file to resolve external references to the dynamic link routines. Ordinary .LIB files resolve external references by supplying the object code that is referenced. The .LIB files built by IMPLIB resolve external references by supplying special records that contain pointers to the dynamic link libraries and entry points. OS/2 links an application (or another dynamic link library) to the dynamic link routines that are called at load time. For additional information see "Module Definition Files" on page 4-62.

For more information about the IMPLIB utility, see the *OS/2 Programmer's Guide*.



IBM FORTRAN/2 subroutines, intrinsic and external functions, and block data subprogram units can be linked into a dynamic link library.

The entry points to these external procedures and the names of common blocks can then be exported for use in an application program.

The procedure for building a dynamic link library and its corresponding import library is to:

1. Compile the subprogram units you intend to link into the dynamic link library.

2. Create a module-definition file that defines the dynamic link library (using the LIBRARY statement) and its exports (using the EXPORTS statement). You can use other module-definition statements to define segment and module characteristics. For more information see "Module Definition Files" on page 4-62. See "Compiler Generated and Library Segment Names" on page 8-18 for information about segments and PUBLIC symbols generated by the compiler and intrinsic functions.

3. Link the subprogram object modules with the /STACK:2 option. (Only a minimum stack is needed since the application program's stack will be used.)

   Make sure the appropriate IBM FORTRAN/2 library (FORTRAN.LIB or FORTRAE.LIB for OS/2 mode) is available as well as any other libraries and import libraries. (For example, be sure to include DOSCALLS.LIB if the subprograms make application program interface calls.)

4. Check the link map. Make sure all desired PUBLIC symbols have been exported, particularly those intrinsic functions also needed by the application. If required, modify the module-definition file to include all required PUBLIC symbols as exports, and then relink.

5. After all the required PUBLIC symbols have been included in the module-definition file, use the IMPLIB utility (part of the OS/2 Toolkit) to generate an import library with the module-definition file as input.

To use the dynamic link library in an application program, link the application, noting the following:

1. The import library created above must be included as one of the libraries.

2. If intrinsic functions are exported, make sure the library search is arranged so that the import library is searched before the IBM FORTRAN/2 library (FORTRAN.LIB or FORTRAE.LIB) is searched.

3. Request a link map when linking.
4. Make sure all desired PUBLIC symbols are imported (review link map). If not, modify the module-definition file to include all required PUBLIC symbols as exports, relink the dynamic link library, rebuild the import library and relink the application.
5. The size of the stack segment (as specified in the link map) is large enough. If not, relink the program, increasing the value of the /STACK option appropriately.

When the application is run, the dynamic link library must be accessible in the search path established by the OS/2 LIBPATH command in the CONFIG.SYS file.

Make sure you restart your system after modifying CONFIG.SYS.

The following example shows how to build and use a dynamic link library.

1. Separate the demonstration program DEMO.FOR into two files: one containing the main program unit (which will be named DEMOMAIN.FOR and used as the application program) and the other containing the subprogram units MYSIN.FOR and FACTRL.FOR (which will be named DEMOSUBS.FOR and used for the dynamic link library).
2. Compile DEMOMAIN.FOR and DEMOSUBS.FOR. (Make sure you use the /N compiler option if you do not have a math coprocessor.
3. Create a module-definition file (called DEMOSUB.DEF) that contains the following:

```
; DYNAMIC MODULE-DEFINITION FOR DEMO SUBROUTINES
  LIBRARY DEMOSUBS
  PROTMODE
  DESCRIPTION 'DEMOSUBS'
  CODE PRELOAD EXECUTEREAD
  DATA PRELOAD READWRITE NONSHARED
  EXPORTS
     MYSIN  @1
```

4. Link the module-definition file:

```
LINK DEMOSUBS /M /STACK:2,,,,DEMOSUBS
```

   Make sure the appropriate libraries are available (in particular FORTRAN.LIB or FORTRAE.LIB for OS/2 mode).

5. Copy DEMOSUBS.DLL to a directory specified in the LIBPATH command of the CONFIG.SYS file, or add the pathname of the directory containing DEMOSUBS.DLL to the LIBPATH command in CONFIG.SYS. If CONFIG.SYS is altered, you must reboot OS/2.

6. Build the import library:

   ```
   IMPLIB DEMOSUBS.LIB DEMOSUBS.DEF
   ```

7. Link the application program:

   ```
   LINK DEMOMAIN /M /STACK:1024,,,DEMOSUBS;
   ```

   Again, make sure the OS/2 mode libraries are accessible.

   Review DEMOMAIN.MAP.

8. Run the application program:

   ```
   DEMOMAIN
   ```

The PUBLIC symbol for a common block may be exported. See "Compiler Generated and Library Segment Names" on page 8-18. This does not, however, imply that a common block can be defined with this name by an application. That is, a common block defined in an application will cause a linker error if there is a PUBLIC symbol imported with the same name as the common block. This is true even if this PUBLIC symbol is defined for a common block in a dynamic link library. The application must access a common block in a dynamic link library through the PUBLIC symbol using a routine written in some other language (such as COPYMEM.ASM).

If multiple dynamic link libraries are used with an application, it may be desirable to link all required intrinsic functions into a separate dynamic link library, which can then be accessed by the other dynamic link libraries and by the application module.

Module Definition Files

A *module-definition file* is an ASCII text file that provides additional information about the OS/2 mode .EXE or .DLL file being built. A module-definition file is required when you link a dynamic link library and is optional when you link an application. The file contains one or more definition statements. Each statement defines some aspect of the application or dynamic link library, such as segment attributes and

functions exported. You can choose any file name and extension for a module-definition file. The default extension is .DEF.

The following sections explain how to create module-definition files for application and dynamic link libraries.

Module Definitions for Applications

A module-definition file for an application is optional. You can use it, for example, to change the stack size or to cancel the default segment characteristics.

The following example shows a module-definition file for an application:

```
;Sample Module-Definition File
NAME
DESCRIPTION  'Sample .DEF file for Application'
CODE         LOADONCALL
STACKSIZE 2048
```

In this example, the NAME statement specifies to the linker that the .EXE file being created is an OS/2 mode application. The CODE statement specifies that code segments are loaded on demand. The STACKSIZE is 2048 bytes.

The first line of the sample module-definition file is a comment. A comment can appear on a line by itself, or on the same line as a definition statement, as long as it appears after the definition. A semicolon must precede a comment.

Module Definitions For Dynamic Link Libraries

A module-definition file for a dynamic link library must contain a LIBRARY statement specifying that the .DLL file being created is a dynamic link library. The file must also contain an EXPORTS statement that lists the functions within the dynamic link library to be exported. Functions in the dynamic link library are not accessible unless they are listed.

The following example shows a module-definition file for a dynamic link library:

```
;Sample Module-Definition File
LIBRARY
DESCRIPTION  'Sample .DEF file for Dynamic Link library'
CODE    LOADONCALL
EXPORTS
         Init  @1
         Start @2
         End   @3
         Load  @4
         Save  @5
```

In this example, the LIBRARY statement specifies to the linker that the .DLL file being created is a dynamic link library. The CODE statement specifies that code segments should be loaded on demand. The EXPORTS statement lists the dynamic link entry points to be exported by name and number.

Dynamic link routines use the stack of their caller.

Module-Definition Statements

The module-definition file defines the contents and system requirements of an OS/2 mode application or dynamic link library. The file contains one or more module statements, each defining a specific attribute of the application, such as its name, the number and type of program segments, and the number and names of exported or imported functions.

The following section describes the module statements relevant to IBM FORTRAN/2:

Statement	Description
CODE	Code segment attributes
DATA	Data segment attributes
DESCRIPTION	One-line description of the module
EXPORTS	Exported functions
IMPORTS	Imported functions
LIBRARY	Dynamic link library name
NAME	Application name
OLD	Preserves export ordinals
PROTMODE	OS/2 mode only
SEGMENTS	Segment attributes per segment name
STACKSIZE	Local stack size in bytes
STUB	DOS mode executable file

Notes:

1. The statement keywords must be uppercase.
2. Any line in the module definition file beginning with a semicolon (;) is a comment and is ignored by the linker and IMPLIB.

CODE

This statement defines the default attributes of all code segments in the application. A code segment is any segment with a class name ending in "code" (uppercase or lowercase).

Format

CODE
[*load-option*] [*execute-option*] [*privilege-option*]

Parameters

The *load-option* is an optional keyword specifying when the segment is to be loaded. It must be one of the following:

LOADONCALL	The segment is loaded when called.
PRELOAD	The segment is loaded immediately.

The default is LOADONCALL, unless you give no CODE statement. In the absence of a CODE statement, the default is PRELOAD.

The *execute-option* specifies whether the segment can be read as well as run. It must be one of the following:

EXECUTEREAD	The segment can be run and read.
EXECUTEONLY	The segment can only be run. (This should not be specified for IBM FORTRAN/2 code segments.)

The default is EXECUTEREAD.

The *privilege option* is an optional keyword specifying if the segment has I/O privilege. It must be one of the following:

IOPL	The segments have I/O privilege.
NOIOPL	The segments do not have I/O privilege.

The default is NOIOPL.

Note: Only one CODE statement is allowed in a module-definition file.

This statement defines the attributes of the data segments of the application. A data segment also contains the stack and heap of the application. A data segment is any segment that is not a code segment. A code is a segment that has a class name ending in "code" (uppercase or lowercase).

Format

DATA [*instance-option*] [*shared-option*] [*write-option*]
[*privilege-option*] [*load-option*]

Parameters

The *instance-option* is an optional keyword that describes sharing of the automatic data segment, which is a group named "DGROUP". It can be any one of the following:

MULTIPLE The automatic data segment is copied for each instance of the module.

SINGLE The automatic data segment is shared by all instances of the module (valid only for dynamic link libraries).

NONE There is no automatic data segment.

The default is MULTIPLE for program modules and SINGLE for dynamic link libraries.

The *shared-option* is an optional keyword specifying the need to have a unique copy of the READWRITE data segments loaded for each process that is using the dynamic link library. Values are:

NONSHARED A unique copy of each READWRITE data segment will be loaded for each process using the dynamic link utility.

SHARED A single copy of each data segment will be loaded. (This should *not* be specified or be the default for IBM FORTRAN/2 data segments.)

The default is NONSHARED for program modules, and SHARED for dynamic link libraries.

DATA

The *write-option* is an optional keyword specifying whether the segment can be written to:

READWRITE	The segment can read from or written to.
READONLY	The segment can only be read from. (This should *not* be specified for IBM FORTRAN/2 data segments.)

The default is READWRITE.

The *privilege option* is an optional keyword specifying if the segment has I/O privilege. (This should *not* be specified for IBM FORTRAN/2 data segments.) It must be one of the following:

IOPL	The segments have I/O privilege.
NOIOPL	The segments do not have I/O privilege.

The default is NOIOPL.

The *load-option* is an optional keyword specifying when the segment is to be loaded. It must be one of the following:

LOADONCALL	The segment is loaded when referred to.
PRELOAD	The segment is loaded immediately.

If a DATA statement is present, the default for the *load-option* is LOADONCALL. In the absence of a DATA statement, the default is PRELOAD.

Note: Only one DATA statement is allowed in the module-definition file.

This statement inserts *text* into the module of the application. It is useful for embedding source control or copyright information.

Format

DESCRIPTION *text*

Parameters

The *text* is one or more ASCII characters. You must enclose the string in single quotation marks.

Example

```
DESCRIPTION 'Template Application'
```

Note: Only one DESCRIPTION statement is allowed in a module-definition file.

EXPORTS

EXPORTS defines the names and characteristics of the functions in the dynamic link library to export to other applications or dynamic link libraries. The EXPORTS keyword marks the beginning of the definitions. Following the EXPORTS keyword are any number of export definitions (up to 3072), each on a separate line.

Format

EXPORTS *exportname* [*ordinal-option* [RESIDENTNAME]] [*iopl-parmwords*]

Parameters

The *exportname* is one or more ASCII characters defining the function name. It has the form:

```
entryname[=internalname]
```

The *entryname* is the name that other applications use to access the exported function. It is a required parameter.

The *internalname* defines the actual name of the function if *entryname* is not the actual name. It is an optional parameter.

The *ordinal-option* defines the ordinal value of the function. It has the form:

@ordinal

where *ordinal* is an integer number specifying the ordinal value of the function. The ordinal value defines the index of the name of the function in the entry table of the dynamic link library. A space must precede the @ character. It is an optional parameter.

RESIDENTNAME indicates that the entry point name of the function should be kept resident in memory. It is an optional parameter and applies only when an *ordinal-option* is specified. Otherwise, name strings are always kept resident. Keeping frequently used entry point name strings resident in memory allows calls to be resolved more rapidly, assuming the call by entry point name rather than ordinal.

The *iopl-parmwords* is an optional numeric value that must be specified for functions which execute with I/O privilege. A function that executes with I/O privilege is allocated a 512-byte stack. When the function is invoked, the number of parameters (words) specified by *iopl-parmwords* are copied from the caller's stack to the new stack. Do not specify this parameter for functions which do not have I/O privilege level.

Only one EXPORTS statement is allowed in a module-definition file.

Example

```
EXPORTS
        SampleRead @1 RESIDENTNAME
        StringIn=str1 @2
        CharTest @3
```

IMPORTS

IMPORTS defines the names and attributes of functions to be imported from existing dynamic link libraries. The IMPORTS keyword marks the beginning of the definitions. Following the IMPORTS keywords are any number of definitions, each on a separate line.

Format

IMPORTS
[*internal-name* =] *module.entry*

Parameters

internal-name is an optional parameter that specifies the name that the application uses to call the function. *internal-name* is one or more ASCII characters. This name must be unique.

If omitted, the name used by the referencing application or dynamic link library must be the same as *entryname*.

module is the name of a dynamic link library containing the function.

entry is the function to import. It is one of the following:

- *entryname* Is the actual name of the function.
- *entryordinal* Is the ordinal value of the function. The ordinal used here is the same as for the entry point in the dynamic link library. If this specified, an *internal-name* must be given.

Only one IMPORTS statement is allowed in a module-definition file.

Example

```
IMPORTS
        Sample.SampleRead
        write2hex=Sample.SampleWrite
        Read.1
```

This statement specifies that the file being created is a dynamic link library. It also specifies the type of library initialization required for the dynamic link library.

Format

LIBRARY [*libraryname*] [*initialization-type*]

Parameters

The *libraryname* is optional. If none is specified, the linker uses the name of the dynamic link library, without any path prefix and extension.

Once a dynamic link library has been loaded, *libraryname* is the name of the library as known by OS/2.

initialization-type is an optional keyword specifying the type of library initialization required by the dynamic link library. This keyword is ignored if a library initialization routine is not defined for the library module. It must be one of the following:

INITGLOBAL The library module initialization routine is called only once when the dynamic link library is initially loaded.

INITINSTANCE The library module initialization routine is called once for each process that gains access to the dynamic link library.

The default is INITGLOBAL.

Remarks

If the LIBRARY statement is omitted, the module is an application module.

Only one LIBRARY statement is permitted. It should be the first statement, if specified.

You cannot specify both LIBRARY and NAME statements in the same module-definition file.

LIBRARY

A default file extension of .DLL will be used for the dynamic link library file.

Example

```
LIBRARY User
```

This statement specifies that the file being created is an application module.

Format

NAME [*modulename*]

Parameters

The *modulename* is optional. If none is specified (or there is no NAME or LIBRARY statement), the linker uses the name of the executable file, without any path prefix and extension. Once an application program module has been loaded *modulename* is the name of the application as known by OS/2.

Remarks

If the NAME statement and LIBRARY statement are omitted, the module is an application module.

Only one NAME statement is permitted. It should be the first statement, if specified.

You cannot specify both NAME and LIBRARY statements in the same module-definition file.

Example

```
NAME Calendar
```

OLD

This statement preserves export ordinals across successive versions of a dynamic link library.

Format

OLD '*libraryname*'

Parameters

libraryname is the name of the dynamic link library to be used for export ordinals. The *libraryname* must be within single quotation marks.

Remarks

Exported names in this library matching exported names in the OLD library will be assigned ordinal values from the OLD library unless:

- The name in the OLD library did not have an assigned ordinal

  or

- An ordinal was explicitly assigned to a name in this library.

Note: If the linker cannot find *filename* in the current directory, it looks in the directories listed in the LIBPATH environment variable.

This statement causes the linker to set the protected-mode-only bit in the file header of the executable or dynamic link library. Without a PROTMODE statement, the OS/2-mode-only bit is not set; this is the default setting.

Format

PROTMODE

Remarks

This statement must be specified for executable files that are dynamic link libraries.

If an application module is to be bound (using BIND) so that it will run in both DOS mode and OS/2 mode, this statement must *not* be specified.

This statement should be specified for application modules that will be run only in OS/2 mode.

Note: IBM FORTRAN/2 uses its own emulator so that the two following paragraphs do not apply to IBM FORTRAN/2 object files.

PROTMODE

If the linker recognizes floating-point instructions in the object module and the protected-mode-only bit is not requested to be set, the linker produces runtime relocations of type OSFIXUP. If you use the resulting executable file as input to the BIND utility, BIND can emulate the floating-point instructions for DOS mode if no coprocessor is present. If the protected-mode-only bit is requested to be set, the linker does not produce OSFIXUP relocations and the BIND utility cannot be used to emulate the floating-point instructions for DOS mode.

For programs that use floating-point instructions, the amount of space in the executable file that contains OSFIXUP relocations can be significant. By using a definition file containing the PROTMODE statement, you can save space in the executable file. Do this only if you intend to run the program only in OS/2 mode.

This statement defines code and data segment attributes on a per segment basis. The parameters specified override the defaults established by the CODE and DATA segments. The SEGMENTS keyword marks the beginning of the definitions. Following the SEGMENTS keyword are any number of segment definitions, each on a separate line.

Format

SEGMENTS[']*segmentname*['] [*class-option*][*segmentflags*]

Parameters

Each segment definition consists of a combination of the following parameters:

segmentname is a character string naming the segment. Optionally, you can enclose the *segmentname* in single quotation marks. If the *segmentname* is CODE or DATA, you must enclose it in single quotation marks to avoid conflict with the CODE and DATA statements. The order of the segment names defines the order of the segments in an executable file.

class-option is an optional keyword specifying the class of the segment:

CLASS *'classname'*

You should always specify a *classname* along with the *segmentname* when using SEGMENTS. If no class is given, the linker assumes a class of CODE. If it finds a segment with a class of ''code'' with a name specified, that segment will be defined. If it does not find that segment then it defines a new segment.

segmentflags is any combination of these options that are described under the CODE and DATA keywords above:

SHARED, NONSHARED The default is NONSHARED.

PRELOAD, LOADONCALL The default is LOADONCALL.

EXECUTEONLY, EXECUTEREAD The default is EXECUTEREAD (for code segments).

READONLY, READWRITE The default is READWRITE (for data segments).

IOPL, NOIOPL The default is NOIOPL.

Only one SEGMENTS statement is allowed in a module-definition file.

Example

```
SEGMENT
TEXT LOADONCALL EXECUTEONLY
SYSDATA READONLY IOPL
```

STACKSIZE

This statement defines the number of bytes needed by the application for its local stack. An application uses the local stack whenever it calls its own functions. A minimum stack size of 4096 bytes is recommended for an application. A stack size of 0 is recommended for a dynamic link library since it will use the stack of the application.

The maximum stack size for DOS mode is 65535 bytes. The maximum stack size for OS/2 mode is 65534 bytes. The minimum stack size accepted by the linker is 0.

See "/STACK" on page 4-40 for the default stack size if the STACKSIZE statement does not occur in the module-definition file.

Format

STACKSIZE *bytes*

Parameters

The *bytes* parameter is an integer number specifying the stack size in bytes.

```
STACKSIZE 4096
```

The above example is equivalent to using the option **/STACK:4096**. If both a STACKSIZE and /STACK option occur, STACKSIZE overrides /STACK.

In general, this statement is not required for use with IBM FORTRAN/2. See "/STACK" on page 4-40.

STUB

The STUB statement adds the DOS mode executable file named in *filename* to the beginning of the OS/2 mode module being created.

Format

STUB *filename*

Parameters

The *filename* is the name of the DOS mode .EXE file to add to the module. The name must have the DOS file name format.

When executed under DOS mode, the DOS mode part of the EXE module (the stub) executed. Under OS/2 mode, the OS/2 mode part of the EXE module is executed.

Only one STUB statement is allowed in a module-definition file.

Do not use this statement when the LIBRARY statement is used.

The Map File

The map file lists the names, load addresses, and lengths of all segments in a program. It also lists the names and load addresses of any groups in the program, the program start address, and error messages, if any. If you use the **/MAP** option in the the linker command line, the map file lists the names and load addresses of all public symbols.

In the map file, you may see names that begin with F@, P@, and other names containing an @ symbol. These names are reserved for internal use. See also "Compiler Generated and Library Segment Names" on page 8-18.

Map File for DOS Mode .EXE Files

The DOS mode segment and group information has the general form of the following example. `Start` and `Stop` are the 20-bit addresses of the first and last byte in the segment. `Length` is the length of the segment in bytes. `Name` is the name of the segment, and `Class` is the class name of the segment.

```
Start     Stop   Length Name          Class
0000H     0012CH 0012DH P@DEMO        CODE
0012DH    00235H 00109H P@MYSIN       CODE
00236H    002C7H 00092H P@FACTAL      CODE
002C8H    00662H 0039BH F@ICODE       CODE
00664H    0598BH 05328H F@RT          CODE
0598CH    0611AH 0078FH F@MALC        CODE
06120H    068F9H 007DAH F@DOS         CODE
068FAH    068FAH 00000H F@@DATA       DATA
06910H    06A56H 00147H LAA@DEMO      F@DATA
06A60H    06A7GH 00020H LAA@MYSIN     F@DATA
06A80H    06A9BH 0001CH LAA@FACTRL    F@DATA
06A9CH    06B43H 000A8H F@IDATA1      F@DATA
06B50H    074EFH 009A0H F@IOCOM       F@DATA
074F0H    074FBH 0000CH F@IOSYSI      F@DATA
07540H    07989H 0044AH STACK         STACK
07990H    07990H 00000H ??SEG

Origin      Group
068F:0      DGROUP

Program entry point at 0000:0000
```

Map File for OS/2 Mode .EXE Files

The OS/2 mode segment and group information has the general form of the following example. `Start` is the address of the first byte in the segment in the form *segment number:offset*. `Length` is the length of the segment in bytes. `Name` is the name of the segment, and `Class` is the class name of the segment.

```
DEMO

Start      Length    Name             Class
0001:0000  0012DH    P@DEMO           CODE
0001:012D  00109H    P@MYSIN          CODE
0001:0236  00092h    P@FACTAL         CODE
0001:02C8  0039BH    F@ICODE          CODE
0001:0000  00000H    F@@DATA          DATA
0002:0000  00147H    LAA@DEMO         F@DATA
0003:0000  00020H    LAA@MYSIN        F@DATA
0004:0000  0001CH    LAA@FACTRL       F@DATA
0005:0000  000A8H    F@IDATA1         F@DATA
0006:0000  0044AH    STACK            STACK
0007:0000  00000H    ??SEG

Origin   Group
0002:0   DGROUP

Program entry point at  0001:0000
```

Public Symbol Listings

If you have specified the **/MAP** option in the the linker command line, the linker adds a public-symbol list to the map file. It presents the symbols twice: once in a alphabetical order (if a number is *not* specified in the **/MAP** option), and then in the order of their load addresses.

DOS Mode Public Symbol Listing

In the public symbol lists, the address `segment:offset` is an address relative to the beginning of the program load module, which starts at 0000:0000.

```
Address          Publics by Name
0000:0000        DEMO
0612:0540        DOSALLOCSEG
0612:04C0        DOSCHGFILEPTR
0612:002B        DOSCLOSE
0612:0272        DOSCREATESALIAS
0612:004B        DOSDELETE
0612:0000        DOSDEVCONFIG
0612:01FF        DOSEXIT
0612:078E        DOSFILELOCKS
0612:0070        DOSGETENV
0612:04F1        DOSGETMACHINEMODE
0612:0114        DOSGETVERSION
0612:0137        DOSOPEN
0612:0208        DOSREAD
0612:0502        DOSREALLOCSEG
0612:029A        DOSSETSIGHANDLER
0612:037E        DOSSETVEC
0612:06E1        DOSSLEEP
0612:023D        DOSWRITE
0066:0004        F@8087
0598:030C        F@ALLOC
0066:4EB6        F@CSF2
0066:4F48        F@CSF3
•
•
•
0066:0033        F@STPV
0066:02A9        F@STW
0023:0006        FACTRL
0612:05DD        FAPIADDSIGCATCH
0612:0666        FAPISUBSIGCATCH
0012:000D        MYSIN
Address          Publics by Name
000:000          DEMO
0000:000B  Abs   F@ERMAL
0012:000D        MYSIN
0023:0006        FACTRL
002C:0008        F@EII
002C:0100        F@EFI
002C:0227        F@SSN3
002C:0243        F@FSN3
•
•
•
074F:0014        F@MAOPR
074F:001C        F@MASPO
074F:001E        F@MASPS
```

OS/2 Mode Public Symbol Listing

The addresses of the public symbols are in *segment:offset* format. The offset is the location of the symbol relative to the start of the segment. The segment is the index into the segment table.

Imp indicates an imported symbol (that is, a reference to a dynamic link library).

```
Address            Publics by Name
0001:0000          DEMO
0000:0000   Imp    F@8087               (FORTRUN.F@8087)
0001:03C0          F@EFI
0001:02C8          F@EII
0000:0000   Imp    F@ERR                (FORTRUN.F@ERR)
0001:0503          F@FSN3
0000:0000   Imp    F@IOISF              (FORTRUN.F@IOISF)
0000:0000   Imp    F@IORSF              (FORTRUN.F@IORSF)
0000:0000   Imp    F@IOSPF              (FORTRUN.F@IOSPF)
0000:0000   Imp    F@IOWEF2             (FORTRUN.F@IOWEF2)
0000:0000   Imp    F@IPGM               (FORTRUN.F@IPGM
0001:0603          F@ML2
0001:04E7          F@SSN3
0000:0000   Imp    F@STPV               (FORTRUN.F@STPV)
0001:0236          FACTRL
0001:012D          MYSIN

Address            Publics by Value

0000:0000   Imp    F@IORSF              (FORTRUN.F@IORSF)
0000:0000   Imp    F@IOISF              (FORTRUN.F@IOISF)
0000:0000   Imp    F@8087               (FORTRUN.F@8087)
0000:0000   Imp    F@IPGM               (FORTRUN.F@IPGM)
0000:0000   Imp    F@IOWEF2             (FORTRUN.F@IOWEF2)
0000:0000   Imp    F@ERR                (FORTRUN.F@ERR)
0000:0000   Imp    F@STPV               (FORTRUN.F@STPV)
0000:0000   Imp    F@IOSPF              (FORTRUN.F@IOSPF)
0001:0000          DEMO
0001:012D          MYSIN
0001:0236          FACTRL
0001:02C8          F@EII
0001:03C0          F@EF1
0001:04E7          F@SSN3
0001:0503          F@FSN3
0001:0603          F@ML2
0001:0662          F@INAME1
```

Chapter 5. Running Your Program

This chapter describes the methods and procedures required to successfully run your program. It also contains information on the following aspects of IBM FORTRAN/2 program execution:

- Runtime routines
- Runtime errors
- STOP and PAUSE statement execution
- Using emulation
- Controlling I/O
- File structure.

Starting Your Program

To start your program, enter the program name (that is, the name you gave your load module) in response to the DOS or OS/2 command prompt. You need not specify an extension, .EXE, with your program name. For example, to run a load module named DEMO.EXE, enter:

```
DEMO
```

In OS/2 mode, the appropriate dynamic link library (FORTRUE.DLL or FORTRUN.DLL) should be accessible. See LIBPATH in Chapter 2, "Installation."

You can also change the maximum record length for data transfer when you start program execution. To do this, enter the /R *n* option on your command line. For example:

```
DEMO /R 2048
```

This command changes the maximum record length from a default of 1024 bytes to 2048 bytes. You can also shorten the maximum record length. Such changes are in effect only during the current program run.

For formatted and unformatted I/O, making n smaller will save space at runtime. Making it larger will allow for larger records.

Note: The maximum value for n is 65514. The minimum value for n is 1.

See "Sample Batch Files" on page 2-21 for a sample batch file to run your program.

Canceling Program Execution

To cancel your program while it is running, simultaneously press the CTRL and BREAK keys. When you do this, the following message appears on your screen:

```
Execution terminated because of CTRL-BREAK
```

Any files opened by your program are closed, and control is returned to DOS or OS/2 (the DOS or OS/2 command prompt reappears on your screen).

IBM FORTRAN/2 Library Routines

The IBM FORTRAN/2 library, FORTRAN.LIB, contains the following types of routines:

- Intrinsic function routines
- I/O runtime routines
- Operating system interface runtime routines
- Debug routines
- Miscellaneous runtime routines.

The IBM FORTRAN/2 library, FORTRAE.LIB, contains the same types of routines as above for use with emulator, plus the emulator itself.

Intrinsic Function Routines

Intrinsic function routines are extracted from one of the IBM FORTRAN/2 libraries when your program requires the use of these functions. See Appendix D, "Intrinsic Functions," in the *IBM FORTRAN/2 Language Reference* manual for a complete list of these functions and how to use them.

I/O Runtime Routines

I/O runtime routines are called by compiler-generated instructions to perform all I/O activity that operates independently of the operating system.

These routines are generally used to transfer and format information internally. When transferring information externally, I/O routines call operating system interface runtime routines.

Under OS/2 mode, these routines are dynamically linked.

Operating System Interface Runtime Routines

Operating system interface runtime routines are called by I/O routines to perform all I/O activity that depends upon DOS or OS/2. These include activities that transfer information to or from externally located files. In turn, these routines call DOS or OS/2 I/O library routines.

Under OS/2 mode, these routines are dynamically linked.

Debug Routines

Debug is only used if programs are compiled with /T option.

Under OS/2 mode, these routines are dynamically linked.

Miscellaneous Runtime Routines

The miscellaneous runtime routines are:

Character string routines. These routines handle processing of character strings.

Error routines. These routines are called from intrinsic function routines, I/O runtime routines, or any other routine. Error routines report execution errors.

STOP and PAUSE routines. These routines are called by compiler-generated code to perform STOP and PAUSE statements.

Under OS/2 mode, these routines are dynamically linked.

Emulator Routines

These routines are only used with the Emulator. The Emulator is required when a FORTRAE.LIB library is used and a program is compiled with the /N option.

Under OS/2 mode, these routines are dynamically linked.

Important: The contents of the IBM FORTRAN/2 libraries are structured to provide efficient program execution. It is recommended that you do not make alterations to the contents or organization of these libraries.

Runtime Errors and Warnings

Runtime errors and warnings might occur during program execution. They are generally the result of one of the following circumstances:

- An erroneous IBM FORTRAN/2 statement
- An illogical or incorrect use of an intrinsic function
- An incorrect or illogical I/O statement detected by IBM FORTRAN/2
- An error detected by DOS or OS/2.

When IBM FORTRAN/2 encounters a runtime error or warning, it displays the appropriate message on your screen (standard error output).

Runtime Message File

All IBM FORTRAN/2 runtime messages are contained in the file named FORTRAN.ERR. This file is installed when you install IBM FORTRAN/2. It should be located in the current directory when you run your IBM FORTRAN/2 programs. If you install FORTRAN.ERR in a different directory, use the SET command to access it. See Appendix A, "Messages" in the *IBM FORTRAN/2 Language Reference* manual for a complete list of messages.

If the program encounters an error but cannot locate the message file, the message number will be printed, but the message itself will not appear. If the program encounters an error and the message file is found but the correct message cannot be found in the file, it will display the following message:

Error in reading error messages file

If this error occurs, program execution ends.

All of the explanations for the messages are listed in Appendix A, "Messages" in the *IBM FORTRAN/2 Language Reference* manual. Also, the contents of FORTRAN.ERR can be printed and used for reference. Thus, it is not necessary to have FORTRAN.ERR available at runtime.

If you wish to use this file at runtime, see Chapter 2, "Installation" for information on installing the runtime error message file.

Trackback Records

Trackback information is automatically kept by IBM FORTRAN/2 and details the sequence of calls which lead to a program error.

Each message is followed by a set of trackback records. The subprogram named in the error record is the last subprogram called in the calling sequence. Each higher-level subprogram call is described on one trackback record. These records are ordered from the lowest to the highest in the calling sequence.

A trackback record has the following format:

Called at *name* + *offset*

name is the name of the program unit issuing the call.

offset is the hexadecimal offset of the calling program unit's relevant compiler-generated CALL instruction. The offset is relative to the start of the program unit's executable code.

Controlling I/O

There are a number of ways to control the devices used for I/O during runtime. Using DOS or OS/2 commands, you can perform the following:

- Accept input from a file other than the one specified in an IBM FORTRAN/2 OPEN or READ statement
- Direct one program's output to become another program's input
- Write to a file other than the one specified in an IBM FORTRAN/2 OPEN, WRITE, or PRINT statement.

To accomplish many of these tasks, you need to connect units to filenames. For a complete explanation of connecting files and units, see Chapter 4, "File Processing" in the *IBM FORTRAN/2 Fundamentals* manual.

Controlling Input

During the life of your program, you might want to accept input from a file or device other than the one specified in your program's OPEN or READ statement. Or, you might want to accept input from a device or file other than the DOS or OS/2 standard input device.

Redirecting the Keyboard

If your program is written to accept input from the keyboard (the READ statement contains "UNIT=*" or "UNIT=5", where unit 5 has not been connected to a file other than standard input), you can still accept input from another file or device without recoding your program. To do this, use the DOS or OS/2 < redirection symbol.

For example, before running the program MYPROG, enter:

```
MYPROG <FILEINP
```

MYPROG is a program which was written to accept keyboard input. FILEINP is a file containing program input. The < symbol disables the input from the keyboard and causes it to be accepted from the file FILEINP throughout the program run. This is applicable for one program run.

Redirecting the Opened File

If you specified a file name in an OPEN statement but want to accept input from another file, use the DOS or OS/2 SET command.

For example, before you run your program, enter:

```
SET OLDFILE.FIL=NEWFILE.FIL
```

This command makes NEWFILE.FIL synonymous with OLDFILE.FIL. This causes program references to OLDFILE.FIL to be interpreted as references to NEWFILE.FIL. This applies for as long as DOS or OS/2 remains running.

After your program runs, enter:

```
SET OLDFILE.FIL=
```

This command breaks the connection between OLDFILE.FIL and NEWFILE.FIL.

Redirecting the READ File

If you do not specify a filename in an OPEN statement (or do not include an OPEN statement at all), IBM FORTRAN/2 creates a filename for you. The filename is based on the unit number specified in the OPEN statement or READ statement. For example, if your OPEN or READ statement specifies "UNIT=8" without specifying a filename, IBM FORTRAN/2 automatically establishes a connection to a file named FORT8. For information about device identifications and preconnected files see Chapter 2, "Statements" in the *IBM FORTRAN/2 Language Reference* manual.

To accept input from a file other than the preconnected file, use the DOS or OS/2 SET command.

For example, before you run your program, enter:

```
SET FORT8=NEWFILE.FIL
```

This command makes NEWFILE.FIL synonymous with FORT8 (assuming the OPEN and READ statements use "UNIT=8"). As a result, all references to unit 8 are interpreted as references to NEWFILE.FIL.

After you run your program, enter:

```
SET FORT8=
```

This command breaks the connection between FORT8 and NEWFILE.FIL.

Controlling Output

During the life of your program, you might want to write to a file or device other than the one specified in your OPEN, WRITE, or PRINT statements. Or, you might want to write to a device or file other than the DOS or OS/2 standard output device.

Redirecting the Screen

If your program is written to send output to your screen (a WRITE statement contains "UNIT=*" or "UNIT=6", where unit 6 has not been connected to a file other than standard output), you can still direct output to another file or device without recoding your program. To do this, use the DOS or OS/2 > redirection symbol.

For example, before running the program MYPROG, enter:

```
MYPROG > FILEOUT
```

When program MYPROG sends output to the screen, it is sent to the file FILEOUT instead. The DOS or OS/2 symbol > causes this redirection to take place. The redirection is in effect for only one program run.

Redirecting the Opened Output File

If you specified a name for your file in an OPEN statement, but want to send output to a different file, use the DOS or OS/2 SET command.

For example, before running the program MYPROG, enter:

```
SET OLDFILE.OUT=NEWFILE.OUT
```

This command makes NEWFILE.OUT synonymous with OLDFILE.OUT. As a result, all references to OLDFILE.OUT are interpreted as references to NEWFILE.OUT.

After running MYPROG, enter:

```
SET OLDFILE.OUT=
```

This command breaks the connection between OLDFILE.OUT and NEWFILE.OUT.

Redirecting the WRITE File

If you do not specify a filename in an OPEN statement (or do not include an OPEN statement at all), IBM FORTRAN/2 creates a filename for you. The filename is based on the unit number specified in the OPEN or WRITE statement. For example, if your OPEN or WRITE statement specifies "UNIT=9" without specifying a filename, IBM FORTRAN/2 automatically establishes a connection to a file named FORT9. For information about device identifications and preconnected files see Chapter 1, "Introduction" in the *IBM FORTRAN/2 Language Reference* manual.

To direct output to a file other than the preconnected file, use the DOS or OS/2 SET command.

For example, before you run your program, enter:

```
SET FORT9=NEWFILE.OUT
```

This command makes NEWFILE.OUT synonymous with FORT9 (assuming the OPEN and WRITE statements use "UNIT=9"). As a result, all references to unit 9 are interpreted as references to NEWFILE.OUT.

After you run your program, enter:

```
SET FORT9=
```

This command breaks the connection between FORT9 and NEWFILE.OUT.

Running Programs Compiled With /N

Programs compiled with the /N compile command option do not require the presence of a math coprocessor. But they must always be linked with the appropriate emulator versions of the libraries (FORTRAE.LIB).

Floating point operations are performed by emulating instructions that would be executed by a math coprocessor. If a math coprocessor is also present, the floating point operations will use it. Programs compiled with the /N option will run much faster on machines that have a math coprocessor. For each set of options and configurations, the following lists the performance order from fastest to slowest:

1. Without the /N and with a math coprocessor
2. With the /N and with a math coprocessor in DOS mode
3. With the /N and with a math coprocessor in OS/2 mode
4. With the /N and without a math coprocessor in DOS mode
5. With the /N and without a math coprocessor in OS/2 mode

In DOS mode with a math coprocessor, the emulator changes each call to the emulator to a jump to the instruction following the call. Therefore, if the sequence of instructions is executed again, a call to the emulator will not occur.

Differences Between Emulation and the Math Coprocessor

For reasons of speed and accuracy as well as software limitations, the following differences exist:

1. All numeric operations are done at maximum precision (64 bits). If less precision is set in the CONTROL register, the emulator ignores this and still does its computations at the maximum precision. If the stack is used to optimize calculations, results may be more precise (and therefore differ) from results that are rounded to less precision after each operation.

2. The precision of the bias exponent in the emulator is 16 bits versus 15 bits in the math coprocessor. This improves the speed of the emulator since it does not have to handle unnormalized numbers internally. Thus there are instances where the emulator will not overflow or underflow when the math coprocessor will. These cases are restricted to numerical values not representable by the math coprocessor.
3. The results produced by the emulator may vary slightly (1 or 2 bits in the low order precision of the 64-bit value) from the math coprocessor for transcendental instructions.
4. The instruction pointer, data pointer, and instruction opcode fields are not supported for FSAVE/FRSTOR instructions. These fields in the memory save area are zeroed when FSAVE and FRSTOR instructions are executed.
5. Unmasked errors do not set the error summary status bit and no interrupt is generated. These are treated the same as masked errors.
6. When the emulator finds a reserved floating point instruction opcode, it issues a message and terminates the program.
7. The emulator cannot be shared between threads of a process. That is, the state of the emulator is not saved and restored when a context switch occurs between threads.

Using Emulation with Assembly Language

The emulator may be called from assembly language. An assembly language program that contains floating point instructions should be modified as follows:

1. Add an

   ```
   EXTRN F@EMUL:FAR
   ```

   (outside of any segments).
2. Before any contiguous sequence of floating point instructions (the sequence may contain FWAIT and NOP instructions), insert

   ```
   CALL F@EMUL
   ```

3. To reduce the number of calls to the emulator, non-floating point instructions which occur in a sequence of floating point instructions should be moved out of the sequence if possible.

Chapter 6. LIB: The Library Manager

The IBM Library Manager/2 (LIB) helps you create, organize, and maintain run-time libraries. Run-time libraries are collections of compiled or assembled functions that provide a common set of useful routines. Any program can call a run-time routine as though the program includes the function. When you link the program with a run-time library file, LINK finds the routine in the library file and resolves the call to the run-time routine.

You create run-time libraries by combining separately-compiled object files into one library file. A .LIB extension identifies a library file, but allows other extensions.

When you incorporate an object file in a library, the object file becomes an object module. LIB makes a distinction between object files and object modules: an *object file* is an independent file but an *object module* is part of a larger library file.

An object file can have a full path name (including a drive designation and a directory path name) and a file name extension (usually .OBJ). Object modules have only a name. For example, B:\RUN\SORT.OBJ is an object file name, but SORT is the name of the corresponding object module.

Overview of LIB Operation: You can perform a number of library management functions with IBM LIB:

- Create a library file
- Delete modules
- Extract a module and place it in a separate object file
- Extract a module and delete it
- Add an object file to a library as a module, or add the contents to a library
- Replace a module in the library file with a new module
- Produce a listing of all public symbols in the library modules.

For each library session, LIB first reads and interprets your commands. It determines whether you are creating a new library or if you are examining or changing an existing library.

LIB processes deletion and extraction commands (if any) first. It does not delete modules from the existing file. Instead, it marks the selected modules for deletion, creates a new library file, and copies only the modules not marked for deletion into the new library file.

Next, LIB processes any addition commands. Like deletions, it does not perform additions on the original library file. Instead, it appends the additional modules to the new library file. (If there were no deletion or extraction commands, LIB creates a new library file in the addition stage by copying the original library file.)

As LIB carries out these commands, it reads the object modules in the library, checks them for validity, and gathers the information necessary to build a library index and a listing file. The linker uses the library index to search the library.

The listing file contains a list of all public symbols in the index and the names of the modules in which they are defined. LIB produces the listing file only if you ask for it during the library session.

LIB never makes changes to the original library; it copies the library and makes changes to the copy. When you end LIB for any reason, you do not lose your original file. It also means that when you run LIB, enough space must be available on your disk for both the original library file and the copy.

When you change a library file, LIB gives you the option of specifying a different name for the file containing the changes. If you use this option, LIB stores the changed library under the name you give, and preserves the original, unchanged version under its own name. If you choose not to give a new name, LIB gives the changed file the original library name, but keeps a backup copy of the original library file. This copy has the extension .BAK instead of .LIB.

The LIB command is easy to use. Its syntax is straightforward, and it prompts you for responses. After you know how LIB works and what its prompts mean, you can use one of the alternate methods of calling LIB, described later in this chapter. These alternative methods let you give LIB commands without waiting for the LIB prompts. A list of LIB error messages is in Appendix A, "Error Messages," of the *IBM FORTRAN/2 Language Reference* manual.

Example

The following command deletes a library module named HEAP from the library file LANG.LIB, then adds a file named HEAP.OBJ as the last module in the library:

```
LIB LANG-HEAP+HEAP;
```

This command can also be given this way, with the same effect:

```
LIB LANG+HEAP-HEAP;
```

This command always performs delete operations before add operations without regard to the order of operations in the command line. This order prevents LIB from flagging the operation as an error when a new version of a module replaces an old version in the library file.

After a library is changed, the command writes the changed file back to the library file LANG.LIB. LANG.BAK is the name of the original backup file of LANG.LIB.

Starting LIB

The library manager, LIB.EXE, is located on the IBM FORTRAN/2 "COMPILER" work diskette or on your fixed disk directory \FORTRAN. If installed on a network, it is found in the directory \APPS\FORTRAN.

You can start the LIB program by using one of the following methods:

- Prompts
- Command Line
- Response File.

The prompt method displays a prompt for each response it needs in the LIB program. See "Prompts for LIB" in this chapter for information on how to use the prompt method. When you understand the LIB prompts and operations, you can use the command line method of running LIB.

The command line method lets you type all commands, options, and file names on the line you use to start LIB. See "Command Line for LIB" in this chapter for information on the command line method.

With the response file method, you create a file that contains all the necessary commands, then tell LIB where to find that file. See "Response File for LIB" in this chapter for information on the response file method.

All of the above methods require that you understand how LIB works and what your responses to its prompts mean. For this reason, we recommend that you allow LIB to prompt you for responses until you are comfortable with its commands and operations.

Prompts for LIB

You start LIB at the DOS prompt by typing **LIB**.

LIB prompts you for the input it needs by displaying the following prompts, one at a time. LIB waits for you to respond to a prompt before it displays the next one.

Library name:
Operations:
List file:
Output library:

The following sections explain the responses you can make to each prompt.

Library Name Prompt

At the **Library name** prompt, give the name of the library file you want. Library file names usually have a .LIB extension. You can omit the extension when you give the library file name because LIB assumes an extension of .LIB. However, if your library file does not have the .LIB extension, include the extension when you give the library file name; otherwise, LIB cannot find the file.

The program allows path names with the library file name, so you can give LIB the path name of a library file in another directory or on another disk.

Because LIB manages only one library file at a time, it allows only one file name in response to this prompt. There is no default response, so LIB produces an error message if you do not give a file name.

If you give the name of a library file that does not exist, LIB displays the prompt:

Library file does not exist. Create?

Your response is **y** if you want to create the library file or **n** if you do not. If you answer **n**, LIB returns control to the DOS prompt.

If you type a library file name and follow it immediately with a semicolon (;), LIB performs only a consistency check on the given library. A consistency check tells you whether all the modules in the library are in usable form. LIB prints a message only if it finds an incorrect object module; no message appears if all modules are intact.

You can also set the library page size following this prompt. See "Setting the Library Page Size" in this chapter for more information.

Operations Prompt

Following the **Operations** prompt, you can type one of the command symbols for manipulating modules (+, −, − +, *, − *) followed immediately by the module name or the object file name. You can specify more than one operation following this prompt, in any order. The default for the **Operations** prompt is no changes.

When you have a large number of modules or files to manipulate (more than can be typed on one line), type an ampersand (**&**) as the last symbol on the line, immediately before pressing Enter. The ampersand must follow a file name; you cannot give an operator as the last character on a line you want to continue. The ampersand causes LIB to repeat the **Operations** prompt, allowing you to specify more operations and names.

The following list describes command symbols and their meanings and uses.

Command Symbol	Meaning and Use
+	The plus sign adds an object file to the library file. Give the name of the object file immediately after the plus sign. You can use path names for object files. LIB supplies the .OBJ extension so you can omit the extension from the object file name. You can also use the plus sign to combine two libraries. When you give a library name after the plus sign, it adds a copy of the contents of the library to the library file it is changing. You must include the .LIB extension when you give a library file name. Otherwise, LIB uses the default .OBJ extension when it looks for the file.
–	The minus sign deletes a module from the library file. Give the name of the module you want to delete immediately after the minus sign. A module name has no path name and no extension.
– +	Type a minus sign followed by a plus sign to replace a module in the library. Give the name of the module you want to replace after the replacement symbol. Module names have no path names and no extensions. To replace a module, LIB first deletes the specified module, then adds to the object file having the same name as the module. The command assumes the object file has an .OBJ extension and resides in the current working directory.
*	Type an asterisk followed by a module name to copy a module from the library file into an object file of the same name. The module remains in the library file. When LIB copies the module to an object file, it adds the .OBJ extension, the drive designation, and path name of the current working directory to the module name to form a complete object file name.

	You cannot override the .OBJ extension, drive designation, or path name given to the object file, but you can later rename the file or copy it to another location.
–*	Use the minus sign followed by an asterisk to move an object module from the library file to an object file. This operation is equivalent to copying the module to an object file, as described above, then deleting the module from the library.

List File Prompt

After the **List file** prompt, you can give a filename for a cross-reference listing file. You can specify a full path name for the listing file to create it outside your current working directory. You can give the listing file any name and any extension. LIB does not supply a default if you omit the extension.

A cross-reference listing file contains two lists. The first is an alphabetical listing of all external (public) symbols in the library. The name of the module to which the symbol name refers comes after that symbol name.

The second list is a list of the modules in the library. Under each module name is an alphabetical listing of the public symbols defined in that module. The default when you omit the response to this prompt is the special file name NUL.LST, which tells LIB not to create a listing file.

Output Library Prompt

After the **Output library** prompt you can give the name of a new library file you want to create with the specified changes. The default is the current library file name; the original, unchanged library name remains the same, but the extension changes to .BAK replacing the .LIB extension. This prompt appears only if you specify changes to the library following the **Operations** prompt.

Selecting Default Responses to Prompts

LIB supplies the default response wherever you omit responses. If you want to suppress the prompts, after any entry but the first, use a single semicolon (;) and then press Enter to select default responses to the remaining prompts. You can use a semicolon with the command line and response file methods of calling LIB.

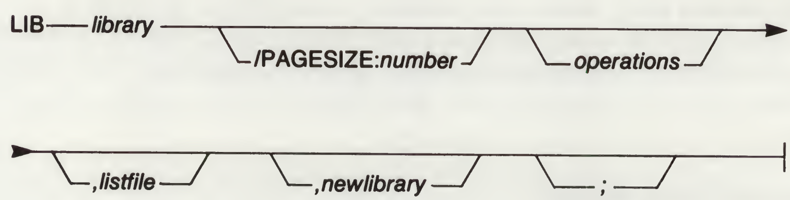
The default response for the **Operations** prompt is no operation. The library file does not change.

The default response for the **List file** prompt is the special filename NUL.LST, which tells LIB not to create a listing file.

The default response for the **Output library** file is the current library name. This prompt appears only if you specify at least one operation following the **Operations** prompt.

Command Line for LIB

The command line method of starting LIB has the following form:



The entries following LIB correspond to the responses to the LIB command prompts.

library — This parameter, with the optional /PAGESIZE:*number* specification, corresponds to the **Library name** prompt. If you want LIB to perform a consistency check on the library, follow the *library* entry with a semicolon (;).

operations — These entries are any of the operations allowed following the **Operations** prompt.

If you want to create a cross-reference listing, you must separate the name of the listing file from the last *operations* entry by a comma. If you give a file name in the new library field, the library name must be separated from the listing file name or the last *operations* entry by a comma.

listfile If specified, LIB creates a listing file with the name.

newlibrary If specified, this is the name of the revised library.

You can use a semicolon after any entry to tell LIB to use the default responses for the remaining entries. The semicolon should be the last character on the command line.

Example

The following example instructs LIB to replace the HEAP module in the library LANG.LIB:

```
LIB LANG-+HEAP;
```

LIB first deletes the HEAP module in the library, then appends the object file HEAP.OBJ as a new module in the library. The semicolon at the end of the command line tells LIB to use the default responses for the remaining prompts. This means that for this command LIB will not create a listing file and LIB will write the changes back to the original library file instead of creating a new library file.

The next example causes LIB to perform a consistency check of the library file C.LIB:

```
LIB C;
```

Does not perform any other action. LIB displays any consistency errors it finds and returns to the operating system level.

The last example tells LIB to perform a consistency check of the library file LANG.LIB, then produces a cross-reference listing file named LCROSS.PUB:

```
LIB LANG,LCROSS.PUB;
```

Response File for LIB

The command to start LIB with a response file has the following form:

LIB *@response-file*

response-file This parameter specifies the name of a response file. The *response-file* name can be qualified with a drive and directory specification to name a response file from a directory other than the current working directory.

Before you use this method, you must set up a response file containing answers to the LIB prompts. This method lets you conduct the library session without typing responses at the keyboard.

A response file has one text line for each prompt. Responses must appear in the same order as the command prompts. Enter responses in the response file the same way you would enter responses on the keyboard.

When you run LIB with a response file, the prompts display with the responses from the response file. If the response file does not contain answers for all the prompts, LIB uses the default responses.

Example

```
SLIBC
+CURSOR+HEAP-HEAP*FOIBLES
CROSSLST
```

This response file causes LIB to delete the module HEAP from the SLIBC.LIB library file, extract the module FOIBLES and place it in an object file named FOIBLES.OBJ. It adds the object files CURSOR.OBJ and HEAP.OBJ as the last two modules in the library. Finally, LIB creates a cross-reference file named CROSSLST.

Extending Lines

If you have many operations to perform during a library session, use the ampersand (&) command symbol to extend the operations line. Give the ampersand symbol after an object module or object file name. Do not put the ampersand between an operations symbol and a name.

If you use the ampersand with the prompt method of calling LIB, the ampersand causes the **Operations** prompt to repeat, letting you type more operations. With the response file method, you can use the ampersand at the end of a line, then continue typing operations on the next line.

Ending the Library Session

At any time, you can use Ctrl+Break to end a library session. If you type an incorrect response, such as a wrong or incorrectly spelled file name or module name, you must press Ctrl+Break to leave LIB. You can then restart the program.

Library Tasks

This section summarizes the library management tasks you can perform with LIB.

Creating a Library File

To create a new library file, give the name of the library file you want to create following the **Library name** prompt. LIB supplies the .LIB extension.

If the name of the new library is the name of an existing file, LIB assumes you want to modify the existing file. When you give the name of a library file that does not currently exist, LIB displays the following prompt:

Library file does not exist. Create?

Type **y** (yes) to create the file; **n** (no) to end the library session.

Note: When you call LIB in such a way that no **Operations** prompt appears, the message above also does not appear. LIB assumes **y** (create the new library) by default. For example,

```
LIB new.lib+obj1;
```

where **new.lib** does not exist, LIB creates the file **new.lib**.

You can specify a page size for the library when you create it. The default page size is 16 bytes. See "Setting the Library Page Size" in this chapter for more information.

After you give the name of the new library file, you can insert object modules in the library by using the add operation (+) following the **Operations** prompt. You can also add the contents of another library. See "Adding Library Modules" and "Combining Libraries" in this chapter for an explanation of these options.

Modifying a Library File

You can change an existing library file by giving the name of the library file following the **Library name** prompt. The **Operations** prompt performs all operations you specify on that library.

LIB lets you keep both the original library file and the newly-changed version. You can do this by giving the name of a new library file following the **Output library** prompt. The library file name changes to the new library file name, while the original library file remains unchanged.

If you do not give a file name following the **Output library** prompt, the changed version of the library file replaces the original library file. LIB saves the original, unchanged library file. The original library file has the extension .BAK instead of .LIB. At the end of the session you have two library files: the changed version with the .LIB extension and the original, unchanged version with the .BAK extension.

Adding Library Modules

Use the plus sign (+) following the **Operations** prompt to add an object module to a library. Give the name of the object file, without the OBJ extension, that you want to add immediately after the plus sign.

LIB removes the drive designation and the extension from the object file specification, leaving only the file name. This becomes the name of the object module in the library. For example, if the object file B:\CURSOR.OBJ is added to a library file, the name of the corresponding object module is CURSOR.

LIB always adds object modules to the end of a library file.

Deleting Library Modules

Use the minus sign (–) following the **Operations** prompt to delete an object module from a library. Give the name of the module you want to delete immediately after the minus sign. A module name has no path name and no extension. It is only a name, such as CURSOR.

Replacing Library Modules

Use a minus sign followed by a plus sign (– +) to replace a module in the library. After the replacement symbol (– +), give the name of the module you want to replace. Module names have no path names and no extensions.

To replace a module, LIB deletes the given module, and adds the object file with the same name as the module. The object file has an .OBJ extension and resides in the current working directory.

Extracting Library Modules

Use an asterisk (*) followed by a module name to copy a module from the library file into an object file of the same name. The module remains in the library file. When LIB copies the module to an object file, it adds the .OBJ extension, the drive designation, and the path name of the current working directory to the module name. This forms a complete object file name. You cannot override the .OBJ extension, drive designation, or path name given to the object file. You can later rename the file or copy it to another location.

Moving Library Modules

Use the minus sign followed by an asterisk (– *) to move an object module from the library file to an object file. This operation is equivalent to copying the module to an object file, then deleting the module from the library.

Combining Libraries

You can add the contents of a library to another library by using the plus sign (+) with a library file name instead of an object file name. Following the **Operations** prompt, give the plus sign (+) and the name of the library with the contents you want to add to the library you are changing. When you use this option, you must include the .LIB extension of the library file name. Otherwise, LIB assumes that the file is an object file and looks for the file with an .OBJ extension.

LIB adds the modules of the library to the end of the library you are changing. The added library still exists as an independent library. LIB copies the modules without deleting them.

After you add the contents of a library or libraries, you can save the new, combined library under a new name by giving a new name following the **Output library** prompt. If you omit the **Output library** response, LIB saves the combined library under the name of the original library you are changing.

Creating a Cross-Reference Listing

Create a cross-reference listing by giving a name for the listing file following the **List file** prompt. If you omit the response to this prompt, LIB uses the special file name NUL.LST. It does not create a listing file.

You can give the listing file any name and any extension. You can specify a full path name, including a drive designation, for the listing file to create it outside your current working directory. LIB does not supply a default if you omit the extension.

A cross-reference listing file contains two lists. The first is an alphabetical listing of all public symbols in the library. The name of the module to which the symbol name refers comes after that symbol name.

The second list is an alphabetical list of the modules in the library. Under each module name is an alphabetical listing of the public symbols to which that module refers.

Performing Consistency Checks

When you give only a library name followed by a semicolon at the **Library name** prompt, LIB performs a consistency check, displaying messages about any errors it finds. It does not make any changes to the library. This option is not usually necessary because LIB checks object files for consistency before adding them to the library.

To produce a cross-reference listing along with a consistency check, use the command line method of calling LIB. Give the library name followed by a semicolon then give the name of the listing file. LIB performs the consistency check and creates the cross-reference listing.

Setting the Library Page Size

The page size of a library affects the alignment of modules stored in the library. Modules in the library are aligned so that they always start at a position that is a multiple of *n* bytes from the beginning of the file. The value of *n* is the page size. The default page size is 16 for a new library or the current page size for an existing library.

Because of the indexing technique LIB uses, a library with a larger page size can hold more modules than a library with a smaller page size. However, for each module in the library, this indexing technique wastes an average of *n*/2 bytes of storage space (where *n* is the page size). In most cases a small page size is advantageous. You should use the smallest page size possible.

To set the library page size, add a page size option after the library file name in response to the the **Library name** prompt:

library-name /PAGESIZE:*n*

The value of *n* is the new page size. It must be a power of 2 and must be between 16 and 32768.

Another consequence of this indexing technique is that the page size determines the maximum possible size of the .LIB file. This limit is 65536 times *number*. For example, **/P:16** means that the .LIB file must be smaller than 1 megabyte (16 times 65536 bytes) in size.

Chapter 7. Debugging Your Program

The IBM FORTRAN/2 Interactive Symbolic Debug program (Debug) helps you resolve runtime errors and increase the overall efficiency of your program. Using Debug, you can perform the following:

- Set breakpoints to check program flow through specific lines of code or to monitor changes in the values of variables
- Step through your program or through individual program units statement by statement
- Examine or change the values of variables, array elements, and arrays
- Control the manner in which you resume execution following a breakpoint
- Trace the flow of individual statements, a range of statements, or subprogram entries and exits
- Display source code and program code mnemonics
- Examine memory locations and registers
- Insert comment lines as part of your Debug input
- Redirect debug input from an alternate source
- Create a diskette or hard-copy record of your Debug session.

To run your IBM FORTRAN/2 program with Debug, complete the following activities:

1. Compile your program using the /T or /NT compiler option.

   Note: Specify the /NT compiler option if you do *not* have a math coprocessor. You may specify other options as desired or appropriate for your configuration, with the exception of the /U option.

2. Link and run the program.
3. When the Debug prompt appears, enter Debug commands. The Debug prompt appears when your program is initially entered, when the STOP statement is encountered (in your program), when the program encounters an error condition, when you press the

attention interrupt keys (Ctrl-Brk), or when your program execution reaches a breakpoint.

These activities are explained in more detail later in this chapter. See "Exercises Using Debug" on page 7-58 for examples of using Debug.

Note: Debug requires approximately 35KB of additional storage for its routines. Before you begin, be sure you have sufficient memory for your program and 35KB for Debug.

Compiling for Debug

To run a program under Debug control, you compile your program with the /T compiler option. See Chapter 3, "Compiling Your Program." This option does the following when it compiles your program units:

- Suppresses code optimization. Code optimization is not possible because you can alter the values of variables and control the order of statement execution during a Debug session.
- Inserts calls to the Debug library. This is performed before each executable statement and each subprogram unit's entry and exit points.
- Creates a set of tables within your program. These tables contain the names and addresses of all variables, labels, and called subprogram units.

Object files compiled with the /T compiler option are larger than files not compiled using this option.

You should also specify the /N option if you do not have a math coprocessor.

Note: The following compiler commands:

```
FORTRAN DEMO /T
```

```
FORTRAN DEMO
```

each produce an object file named DEMO OBJ. To avoid overwriting files, it is recommended that you copy and rename your

Debug object files or perform Debug compilation under a different directory.

Compiling the Main Program Unit

To debug a main program unit, enter the name of the source file that contains the main program unit and the /T compiler option.

For example, to compile a main program unit called MAIN contained in the source file DEMO, enter the following:

```
FORTRAN DEMO /T
```

or, if you do *not* have a math coprocessor:

```
FORTRAN DEMO /NT
```

This compiles the program MAIN, and any other programs in the source file, under Debug control without code optimization.

Compiling Subprogram Units

To debug a specific subprogram unit, you must compile both the main program unit and the subprogram unit using the /T compiler option. Consider the following program structure:

```
        MAIN                Main Program Unit
SUBA           SUBB         Subprogram Units
```

For example, assume MAIN, SUBA, and SUBB are each in a different file. Enter the following compile commands in any order:

```
FORTRAN MAIN /T

FORTRAN SUBA /T

FORTRAN SUBB /T
```

Now both SUBA and SUBB can be executed under Debug control.

Specify the /N option in the above if you do not have a math coprocessor.

Compiling Nested Subprogram Units

To debug a specific nested subprogram unit, you must compile the main program unit and that subprogram unit using the /T compiler option. You may want to compile all subprogram units in the chain of calls (until the target subprogram is encountered) using the /T compiler option. If you don't, you cannot refer to anything in the nested subprogram until you have entered it for the first time.

For example, assume the following program structure exists and that each program is in a different source file:

```
DEMO                 Main Program Unit
MYSIN                Subprogram
FACTOR               Subprogram (called by MYSIN)
```

To debug subprogram unit FACTOR, enter the following compile commands in any order:

```
FORTRAN DEMO /T

FORTRAN MYSIN /T

FORTRAN FACTOR /T
```

Specify the /N option in the above if you do not have a math coprocessor.

Linking for Debug

There are no special requirements for linking a program for Debug. The /T compiler option causes the compiler to encode information in the object files that causes Debug routines to be included in the link.

See Chapter 4, "LINK: The Object Linker" for details on linking.

Setting Up Debug Devices

You can use the Debug default devices or set up your own configuration for accepting Debug commands and displaying output.

The Debug device defaults for I/O are:

- Commands are input from the keyboard (the DOS or OS/2 input device, CON). Redirection and piping do not affect this.
- Responses are displayed on the screen (the DOS or OS/2 output device, CON). Redirection and piping do not affect this.
- Log records of the Debug session are recorded to a disk file DEBUGLOG in the current directory.

Changing Devices

You can change the devices used for I/O during your Debug session using the DOS or OS/2 SET command. Devices can be changed to:

- Accept Debug commands from a file or device other than your keyboard
- Direct Debug responses to a file or device other than your screen
- Write the Debug session log to a file or device other than the Debug log file.

To use the SET command, you must know the names of the logical devices. These names are:

- DEBUGIN - keyboard
- DEBUGOUT - screen
- DEBUGLOG - log file

For example, to send the log file directly to your printer, enter the following before you run your program:

```
SET DEBUGLOG=LPT1
```

This sets the log file, DEBUGLOG, to your printer, LPT1.

Note: See "LOG Command" on page 7-35 for information about how to start and stop logging.

To enter Debug commands from a disk file (called DCNTRL, in the example) instead of from your keyboard, enter the following before you run your program:

```
SET DEBUGIN=DCNTRL
```

The Debug command, INPUT, provides you with an alternate method. Using this command you can switch input from the console keyboard to a file, from a file to another file, or from a file to the console keyboard. See "INPUT Command: I" on page 7-29.

To direct debug responses to a disk file called DOUT, instead of to your screen, enter the following before you run your program:

```
SET DEBUGOUT=DOUT
```

Note: See the *DOS Reference* manual or *OS/2 User's Reference* manual for information on device names.

Debug Messages and Help Files

Make sure the Debug messages and help files (FORTRAN.DER and FORTDBG.HLP) can be accessed by placing them in the current directory, or by using the DOS or OS/2 SET command. If they cannot be found only the message number is displayed and the help facility cannot be used. Example:

```
SET FORTRAN.DER=C:\FORTRAN\FORTRAN.DER
SET FORTDBG.HLP=C:\FORTRAN\FORTDBG.HLP
```

Starting Debug

To start Debug, run your program. Debug stops at the first statement of the program and displays the following message:

```
IBM FORTRAN/2 Interactive Symbolic Debug
Version 1.00
(C)Copyright IBM Corp 1984, 1987
(C)Copyright Ryan-McFarland Corp 1984, 1987

INITIAL ENTRY IN program name
```

The *program name* entry is the name of the program unit you ran. The following prompt:

```
-=>
```

tells you to enter a Debug command.

Referencing Statements, Variables, and Program Units

At times during a Debug session, you might want to perform a Debug command on a specific statement, variable, or program unit. The following sections describe the techniques used to reference a statement, variable, or program unit.

Statements

There are three methods to refer to a specific statement. Each type of reference is used in conjunction with your selected Debug command.

Note: All statement references must refer to the initial line of an executable IBM FORTRAN/2 statement, except in the SOURCE command which can refer to any line, including comment lines.

Line Number

To refer to a specific statement's line number (as it is listed on your program listing), enter the line number. For example, to reference line 10, enter (along with your selected Debug command):

```
10
```

Statement Label

To refer to a specific statement's label, enter a slash (/) followed by the label. For example, to reference label 30, enter (along with your selected Debug command):

```
/30
```

Only labels of executable statements may be used. Labels on non-executable statements may not be used.

Offset

To refer to a specific statement located at an offset (in lines) from a statement label, enter the label, an offset direction of either minus (−) or plus (+), and the number of lines to offset. For example:

/30 + 2   Refers to the second line after label 30

/30 − 2   Refers to the second line before label 30

Variables

To refer to a variable within any of the program units under Debug control, enter the variable name used in the program unit (along with your selected Debug command).

Note: The variable's name cannot contain embedded spaces. Both lowercase and uppercase letters are acceptable. (Lowercase is converted to uppercase.)

When you enter array elements, check that the number of subscripts and values match the array's declared dimensions. Each subscript must be a scalar integer variable or an integer constant. For example:

```
A(1,2,L)
B(I,2)
C(K)
```

If the variable is a subscripted array element, the number of subscripts and their values are checked for range. If the array is an adjustable or assumed-size array, only the number of subscripts is checked for validity. Subscripts cannot contain arithmetic expressions; they must be integer constants or integer variable names. You cannot reference a dummy argument that does not exist in the currently executing program.

Note: You cannot reference an adjustably dimensioned array element. Also, you cannot reference an entire adjustably dimensioned or assumed size array. You cannot reference a dummy argument that does not exist in the currently executing program unit.

Qualifying a Reference

When you refer to a statement or variable from within a Debug command, you must make sure that Debug knows which unit you are referring to. To this end, the following terms are defined:

Executing Program Unit. This is the program unit where program execution has been suspended. You cannot alter this; it is used by Debug to respond to commands that resume execution.

Qualified Program Unit. This is the program unit to which your reference applies unless (as explained in the next paragraph) you explicitly qualify another program unit. At the time you receive control after executing any part of your program, your Executing Program Unit and Qualified Program Unit are identical. However, using the QUALIFY Command you can change your Qualified Program Unit.

Explicitly Qualified Program Unit. To refer to a statement or variable outside your Qualified Program Unit, precede the statement or label reference with the name of the new program unit, followed by a period (.).

To refer to the main program unit, enter the main program unit's name (the name specified in the PROGRAM statement). When the PROGRAM statement is not used or no program name is specified, enter the program unit default name, MAIN.

To refer to a subprogram, enter the name specified in the selected subprogram's SUBROUTINE or FUNCTION statement. You cannot refer to a common block name or to a block data subprogram.

For instance, if your Qualified Program Unit is MAIN, but you wish to refer to statement label 25 within SUBA, enter the following (along with the Debug Command):

```
SUBA./25
```

You have now performed a Debug function (for instance, set a breakpoint) at label 25 in SUBA. Your Qualified Program Unit is still MAIN.

You can refer to statements, lines, or labels in a number of program units with one Debug command. To do so, enter each reference separated by a comma (,). Enclose the entire list of references in parentheses.

For example, to refer to line 10 in program unit MAIN (the qualified program unit), and to label 25 of SUBA, enter the following (along with your selected Debug command):

```
(10,SUBA./25)
```

In the example, the first reference to line 10 does not require the name of the program unit because it refers to the qualified program unit (MAIN).

Stop Statement

Debug stops running your program when it encounters the STOP statement. An END or RETURN statement in a main program also generates a STOP; in the rest of this discussion STOP also includes such END and RETURN statements. When Debug encounters a STOP statement, the following message is displayed:

```
STOP STATEMENT AT line[/label]. IN program unit
```

The *line* entry is the program listing line number containing the STOP statement. The *label* entry is the statement label that corresponds to the line number, *line*.

When the STOP message is displayed, you can enter a Debug command to continue your Debug session or enter one of the commands that resume execution of your program and return control to DOS or OS/2. The commands you use to resume execution are discussed later in this chapter.

See "END Command" on page 7-23 for information on ending a Debug session and returning control to the operating system.

Attention Interrupts

You can interrupt your program or a Debug command by using an *attention interrupt*. An attention interrupt is caused when the Ctrl and Break keys are simultaneously pressed. For example, you can use an attention interrupt to stop executing an infinite DO-loop.

Note: It is recommended that you enter the DOS or OS/2 BREAK ON command before you begin a Debug session. If input is expected in a Debug session, the Ctrl-Break key will not be honored until after the input is entered. See the *DOS Reference* manual or *OS/2 User's Reference* manual for information about the BREAK command.

When you interrupt execution of your program during a Debug session, Debug stops executing your program at the next executable statement that was compiled with the /T option and displays the following message:

```
ATTENTION INTERRUPT AT line[/label] IN program unit
```

The *line* entry is the line number (in the program listing) of the next executable statement Debug would have executed. The *label* entry is the statement label, if any, that corresponds to the line number, *line*.

The message is followed by the Debug prompt, which allows you to enter another Debug command.

When you interrupt the execution of a Debug output command (for example, to stop listing an array), Debug stops listing the information and displays the following message:

```
ATTENTION INTERRUPT HAS SUSPENDED DEBUG COMMAND
```

When the message displays and the Debug prompt appears, you can press the Enter key to continue your Debug session. Pressing the Enter key causes Debug to continue listing the information. If you do not wish to continue the interrupted command, you can enter another Debug command. The interrupted Debug command is canceled and your new Debug command is performed.

Comment Lines

Debug allows you to enter comment lines during your Debug Session; these comment lines are written to the Debug Session Log and provide additional documentation of your debug activity. To place comments into the log, enter an asterisk as the first non-blank character of a line, followed by the comment.

Comment lines may also be placed in command input files in order to document the file.

Using Debug Efficiently

The following are some tips on how to use Debug efficiently:

- For any program you plan to debug, print a source listing during compilation (that is, request both the /T and /L compiler options and redirect output to your printer). You can refer to this listing during debugging.
- Make your Debug commands as brief as possible. Most of Debug's keywords have one or two-character abbreviations.
- Try to find the program unit within a given load module that contains the bug you are trying to resolve. If the program unit can be located, compile a subset of the full load module using the /T compiler option.
- To reexecute a part of the program unit being debugged, use the GO command with a statement reference. This way, you can execute certain portions of your program unit without having to load the program again.
- You can mix NEXT and STEP commands to step through portions of a multiple-unit program.

  For example, assume your program contains three program units. You want to step through the main program unit and the first subroutine, but not the second. In this case, you use the STEP command for the first subroutine CALL, and the NEXT command for the second.

Debug Considerations

The following circumstances should be treated with caution:

- When executing conditional commands, such as WHEN, keep in mind that floating-point numbers are at times inaccurate by a few bits. As a result, you should not use the relational operators .EQ. and .NE. with floating point operands.
- Be careful when repeatedly executing a portion of your code using the GO command.

  Jumping into the middle of DO or IF constructs using the GO command (and a statement reference) causes unpredictable results and even program execution exceptions.
- If a variable is defined in a Type statement (logical, character, integer, real, double precision, or complex), but never used elsewhere in your program, no storage is reserved for it in Debug's variable table. Thus. if you refer to it, an error will be generated. If you are going to refer to a defined variable during a Debug session, make sure it has been initialized.

Debug Commands

You can use the full set of Debug commands to perform the following functions:

- Change the qualified program unit: QUALIFY command
- Activate, deactivate, and list breakpoints: AT, WHEN, and LISTBRKS commands
- Examine and change the values of variables: LIST and SET commands
- Resume or end a Debug session: RUN, GO, STEP, NEXT, and END commands
- Trace your program: TRACE and ENTRY commands
- Create a record of your Debug session: LOG command
- Provide assistance: WHERE and HELP commands.

- Display memory, registers and flags: DISPLAY and REGISTERS commands.
- Display source statements and corresponding object code: SOURCE and MACHINE commands.
- Redirect Debug input: INPUT command.

The majority of the Debug commands have an abbreviated form you can use when entering the command. These are shown at the top of each Debug command description page.

Purpose

Activates or deactivates breakpoints within program units or subprograms compiled under the /T compiler option.

AT Command References by Statement

Format.
AT *statement reference* [OFF]
AT (*statement reference*[,*statement reference*]...)[OFF]
AT * OFF

statement reference	Can be by any reference technique (line, label, or offset).
OFF	Deactivates a breakpoint and makes room for new breakpoints to be added.
asterisk (*)	Is used with the OFF option and deactivates all breakpoints within the qualified program unit.

Example

In the following examples, assume that your qualified program unit MAIN and two subprogram units (SUBA and SUBB) have been compiled with the Debug compiler option.

```
AT 10
```

sets a breakpoint at line 10 in MAIN.

```
AT (12,/15,/25+2)
```

sets breakpoints at line 12, label 15, and two lines after label 25 in MAIN.

```
AT (13,/25,SUBA.5)
```

sets breakpoints at line 13 and label 25 in MAIN, and at line 5 in subprogram unit SUBA.

AT Command: A

Remarks

When the executing program unit encounters a breakpoint, it stops *before* it executes the statement and displays a message in the following format:

```
AT: line number[/label] IN program unit
```

The first of the three examples listed above causes the following message to display when Debug reaches line 10 in MAIN:

```
AT:  10 IN MAIN
```

In the second example, when Debug reaches statement label 15, it displays the following message:

```
AT: 26/15 IN MAIN
```

For this example, label 15 is assumed to be defined on line 26.

Using the OFF option, you can deactivate breakpoints (and therefore make room for new breakpoints).

You can use the statement references to deactivate breakpoints in the same way you use them to set breakpoints. The statement reference is a line number, label, or offset. To deactivate the list of breakpoints set in the examples, enter:

```
AT 10 OFF

AT (12,/15,/25+2) OFF

AT (13,/25,SUBA.5) OFF
```

The asterisk deactivates all breakpoints within the qualified program unit (but not those set in program units outside the qualified program unit). For example, if you are currently qualified in MAIN:

```
AT * OFF
```

deactivates all breakpoints in MAIN.

AT Command References by Subprogram Unit

You can activate breakpoints at every entry and exit point of one subprogram unit, multiple subprogram units, or all subprogram units.

Format

AT *subprogram reference* [OFF]
AT (*subprogram reference*[,*subprogram reference*]...) [OFF]
AT /SUBALL [OFF]

subprogram reference	Is the name used in the SUBROUTINE or FUNCTION statement of your subprogram unit (although breakpoints are set at *every* entry and exit point to that subprogram).
/SUBALL	Sets a breakpoint at every subprogram entry and exit point for every subprogram unit under Debug control. (/SUBALL can be abbreviated /S).
OFF	Deactivates a breakpoint and makes room for new breakpoints to be added.

Example

```
AT SUBA
```

sets breakpoints at the entry and exit points of SUBA.

```
AT (SUBA,SUBC,SUBG)
```

sets breakpoints at the entry and exit points of the three subprogram units in the list.

```
AT /S
```

sets breakpoints at the entry and exit points of every subprogram unit currently under Debug control.

Remarks

When Debug encounters a subprogram entry breakpoint, it enters the subprogram unit, stops, and displays a message in the following format:

AT: *entry name* IN *program unit* (ENTRY)

The *entry name* is the name of the entry point reached by Debug. This is the name assigned in a SUBROUTINE or FUNCTION statement, or a name assigned to a secondary entry point (through an ENTRY statement) within your program.

The *program unit* is the name of the subprogram unit entered.

For example, if SUBA contains two entry points, one named SUBA and the other named SECOND, the following messages are displayed when each entry point is reached:

```
AT: SUBA IN SUBA (ENTRY)

AT: SECOND IN SUBA (ENTRY)
```

The subprogram reference sets breakpoints at all exit points. When an exit point is reached (that is, a RETURN statement is about to be executed), Debug displays a message in the following format:

AT: *line number*[/*label*] IN *program unit* (EXIT)

The *line number* is the line number of the RETURN statement. The *label* corresponds to the statement label (when applicable).The *program unit* is the program unit containing the line number.

For example, if the exit point of SUBA is at line 189, statement label 90, Debug displays:

```
AT: 189/90 IN SUBA (EXIT)
```

Subprogram unit breakpoints are deactivated in the same way statement breakpoints are deactivated (the OFF option is used). To deactivate the breakpoints set in the previous examples, enter:

```
AT SUBA OFF

AT (SUBA,SUBC,SUBG) OFF

AT /S OFF
```

Note: Setting breakpoints with /SUBALL applies only to subprogram units not already set with breakpoints. This is especially important when using /SUBALL OFF. For example, if you have set breakpoints at SUBA and SUBB with the AT command:

```
AT (SUBA,SUBB)
```

and then later set breakpoints at all subprogram units:

```
AT /SUBALL OFF
```

all breakpoints are deactivated except for SUBA and SUBB. Because these two are activated independently of the /SUBALL command, they can only be deactivated in the same manner:

```
AT (SUBA,SUBB) OFF
```

DISPLAY Command: D

Purpose

Displays memory in hexadecimal and character format.

Format

DISPLAY *memory reference*[,offset]

DISPLAY *memory reference,*L *len*

DISPLAY [L *len*]

Remarks

memory reference has the format:

`[DS:|ES:|SS:|CS:|hex:]offset`

and defines the starting address (*selector:offset* format) to be displayed. The value of the designated register or the hex value specifies the segment selector. If memory reference is not specified, the display begins at the ending address of the previous DISPLAY command. If there was no previous DISPLAY command, the display begins at the default location of DS:0000.

If an offset is given without a segment register, and SP or BP is used, then the default is SS.

If an offset is given without a segment register and IP is used, then the default is CS.

Otherwise the default is DS.

offset has the format:

`reg|hex [ +|- hex]`

and defines the beginning or ending address offset. The segment register is the same as for the memory reference.

len has the format:

`reg|hex`

and defines the number of bytes to be displayed. Spaces separating L from *len* are optional and may be omitted.

reg has the format:

`AX|BX|CX|DX|SP|BP|SI|DI||IP`

and specifies that the value in the register should be used in the offset or length.

hex is a hexadecimal value (1 to 4 digits) to be used in the segment selector, offset or length. It may have additional leading zeros, which are ignored.

If neither an ending offset nor a length is specified, a length equal to the number of bytes required to fill a line is used.

A carriage return entered after a DISPLAY Command is treated as a DISPLAY command without arguments.

If length is specified as 0, 64KB (65536) is used for length. If the ending offset is less than starting offset, a length of 1 is used instead.

If the specified or default length will cross a segment boundary, the length is reduced until that condition does not arise.

If more lines are to be displayed than can fit on the screen, a message - - **M O R E** - - is displayed. The display pauses until the Enter key is pressed.

Data is displayed as follows:

```
ssss:hhhØ  xx xx xx xx xx xx xx xx-xx xx xx xx xx xx xx xx  aaaaaaaaaaaaaaaa
```

ssss Is the segment selector.

hhh0 Is the segment offset (always a multiple of 16).

xx Is the value of a byte in hexadecimal (or two spaces if the corresponding address is outside the range to be displayed).

– Separates the first eight bytes from the last eight bytes (the – is omitted if both the 8th and 9th bytes are outside the range to be displayed).

a Is the character corresponding to the byte. A character prints as a period if it cannot be displayed. If the corresponding address is outside the range to be displayed, a space is displayed.

END Command

Purpose
Ends a Debug session.

Format
END

Remarks

The message:

```
*** - = > THE DEBUG RUN HAS FINISHED < = - ***
```

is displayed, all files are closed, and control is returned to the operating system.

ENTRY Command: E

Purpose

Traces entries to and exits from all functions and subroutines.

Format

ENTRY [PARM|OFF]

PARM causes the values of dummy arguments of the entry into the subprogram unit to be listed when the subprogram is entered.

OFF stops subprogram entry and exit tracing.

Remarks

Debug displays the following message when a subprogram unit is entered:

ENTRY: *program unit* ENTERED AT *entry name*

The *program unit* is the name of the subprogram unit entered.

The *entry name* is the name of the entry point within that program unit.

Debug displays the following message when a subprogram unit is exited:

ENTRY: RETURN FROM *program unit*

The *program unit* is the name of the subroutine or function exited.

Examples

```
ENTRY: SUBA ENTERED AT SUBA
```

shows that the primary entry point has been entered in subprogram SUBA.

```
ENTRY: SUBA ENTERED AT SECOND
```

shows that a secondary entry point has been entered in subprogram SUBA.

```
ENTRY: RETURN FROM SUBA
```

shows that the RETURN statement within subprogram SUBA has been encountered and the trace exited this subprogram.

Debug displays a list of the dummy argument values when the PARM option is used. This list follows the message indicating that the program unit has been entered.

When the dummy argument is a variable, the message displays:

argument name = value

The *argument name* is the name of the dummy argument.

The *value* is the value contained in the argument name.

When the argument is an array, the message displays:

argument name = value-1, value-2, value-3...

For arrays, only the first 10 array elements are displayed.

```
ALPHA = 500   -2000  45   123456
```

If there are no dummy arguments for the entry into the subprogram unit, Debug displays the following message:

```
NO DUMMY ARGUMENTS
```

GO Command: G

Purpose
Resumes executing your program under Debug control from the current breakpoint to the next breakpoint or STOP statement.

Format
GO [*statement reference*]

statement reference	is the line number or label of a statement where execution is to resume. When the *statement reference* is not used, execution resumes from the current breakpoint.

Examples

```
GO
```

resumes execution from the current breakpoint.

```
GO /1Ø
```

resumes execution at label 10.

```
GO 2Ø
```

resumes execution at line 20.

```
GO /3Ø-2
```

resumes execution two lines before label 30.

Remarks

Another Debug Command will be accepted when one of the following occurs:

- The program runs to completion and terminates on a STOP statement.
- The program encounters an error condition and terminates abnormally.
- A breakpoint (set by the AT Command) is encountered.

- A WHEN condition (set by the WHEN Command) is satisfied.
- An Attention Interrupt is requested from the keyboard.

Debug always displays where it has stopped (program unit and line number) and what caused it to suspend execution.

The limitations on the use of the GO follow closely the limitations on the use of the FORTRAN GO TO statement:

- You cannot branch into a DO-loop.
- You cannot branch into an IF-block.
- You cannot branch out of your executing program unit.

Violating these rules causes unpredictable results.

If a GO command specifies the line number or label of a statement for which a breakpoint is set, the breakpoint occurs immediately.

A GO command specifying the line number or statement label where the program is currently stopped is the same as a GO command without a statement reference.

HELP Command: H

Purpose
Displays information on a number of requested topics.

Format
HELP [*topic*]

topic is the name of a Debug command or debug error code.

Example: HELP MACHINE

Displays the syntax for, and provides a brief functional description of, the MACHINE Command.

```
HELP DBØØØ5
```

Displays the explanation and recovery action (if applicable) for the Debug error code DB0005.

Remarks
If a command name, or its abbreviation, is given, for the topic, the command format and a brief description are displayed.

If an error code is given, the error message, explanation, and suggested user response are displayed.

If no topic (or USEHELP) is specified, a brief explanation of how to use help and a list of the commands are displayed.

Purpose

Switches the input from the console keyboard to a file, from a file to another file, or from a file to the console keyboard.

Format

INPUT [*filename*]

filename Is an optional drive, path, filename and extension. There is no default extension.

Example

```
INPUT
```

If read from the console keyboard, directs Debug to resume reading the current command input file. If read from a file, the command switches reading to the console keyboard.

```
INPUT C:\USR\JACK\CHK1.DCM
```

Makes CHK1.DCM in the specified directory the current command input file and begins reading commands.

Remarks

If a filename is not specified when the command is entered at the console keyboard, command input is switched back to the current input file. If all the commands have been read from the current input file, or if there is no current input file, the command is ignored (that is, input continues to be read from the console keyboard).

If a filename is not specified when the command is read from the input file, command input is switched back to the console keyboard. The position in the input file is remembered so that reading from the file can be resumed as described in the previous paragraph.

If a filename is specified, the current input file is forgotten, the specified file becomes the current input file and commands are read from this file.

INPUT Command: I

If the last line of the command file does not specify another file to be read, input is switched to the console and there is no current input file.

Initially, the current input file is the file designated by the value of DEBUGIN if it occurs in the DOS or OS/2 environment because of the SET command. That is, Debug begins execution as if the first command entered was

```
INPUT DEBUGIN
```

If DEBUGIN is not set in the environment, or the file designated as the value of DEBUGIN is empty or does not exist, commands are read from the console. Note that the DOS or OS/2 SET command can be used before entering the program to be debugged to designate a command file. For example,

```
SET DEBUGIN=STRTUP.DCM
```

Purpose
Displays the values of program variables.

Format
LIST (*variable reference*[,*variable reference*]...)
LIST *

variable reference	Can be a scalar variable, array element, or array. You can enter as many variables in a single LIST command as can fit on one command line.
asterisk (*)	Lists all variables in the current qualified program unit.

Example

```
LIST ALPHA
LIST (I,J,A(I,J))
```

Remarks
Scalar variables are displayed according to their data types in the following way:

variable = value

Examples

```
ALPHA = 2.10000000E+15
I = 1
COMPLEX = ( 5.00000000E+00,-6.67000000E-10)
LOGICAL = T
CHARACTER = CHARACTER
```

Array elements are listed in the following format:

array name (subscript) = value

Example

```
A(1,2) = 1.00000000E+04
```

An entire array is listed in the following format:

array name = value list

The number of values displayed on each line depends on the type of array, as shown by the following table:

ARRAY ELEMENT TYPE	NUMBER OF VALUES LISTED PER LINE
CHARACTER	1
COMPLEX*16	1
REAL*8	2
COMPLEX	2
REAL*4	4
INTEGER*2	4
INTEGER*4	4
LOGICAL*1	24
LOGICAL*4	24

Examples

```
ALPHA =
 2.00000000E+02  1.00000000E+00  2.76150000E-03  3.71689000E+25
-5.90000000E-03 -1.11767000E-52  5.76500000E+04  4.89325000E-02
```

Array values are displayed in their own type format, starting at the beginning of the array and proceeding sequentially, with the leftmost subscript varying most rapidly. The listing of entire assumed size or adjustable dimensioned arrays is not supported.

You can list common block variables, but the view of COMMON is as declared in the qualified program unit. For example, if the current qualified program unit does not declare a common block, variables in the block cannot be listed. You must QUALIFY to a program unit declaring the common block.

You can list dummy arguments, providing they exist in the currently executing program unit. If you QUALIFY to a subprogram which is not active and list its dummy arguments, an error message is displayed.

Purpose
Displays all current breakpoints and WHEN conditions.

Format
LISTBRKS

Remarks
All current breakpoints and WHEN conditions are displayed in the following format and order:

CURRENT BREAKPOINTS
..... /SUBALL
..... (ENTRY/EXIT) IN *program unit*
..... *line*[*/label*] IN *program unit*

CURRENT WHEN CONDITIONS
condition number [ON|OFF]..... QUALIFIED IN *unit*
condition

The *program unit* is the name of the program unit containing the breakpoint.

The *line* is the line number that triggers the breakpoint. The label */label*, is displayed when applicable.

The *condition number* identifies a specific WHEN condition.

The *unit* is the program unit qualified at the time the WHEN Command was entered. All variables appearing in the condition are qualified to this unit unless explicitly qualified elsewhere.

The *condition* is the condition that triggers the WHEN breakpoint.

If no breakpoints or WHEN conditions are set, the following message displays:

```
CURRENT BREAKPOINTS
  NO BREAKPOINTS ARE SET
CURRENT WHEN CONDITIONS
  NO WHEN CONDITIONS ARE SET
```

Examples

```
CURRENT BREAKPOINTS
..... /SUBALL
..... 1Ø IN MAIN
..... 26/15 IN MAIN
..... 36/25 IN MAIN
..... (ENTRY/EXIT) IN SUBA
..... 5 IN SUBA
..... 25/7 IN SUBA
..... (ENTRY/EXIT) IN SUBB
..... 3 IN SUBC

CURRENT WHEN CONDITIONS
1 ON.....QUALIFIED IN MAIN
         @TOTAL
2 ON.....QUALIFIED SUBA
         @TOTAL1
3 ON.....QUALIFIED IN MAIN
         LOGVAR
4 ON.....QUALIFIED IN SUBB
         (.NOT. LOGVAR1)
5 OFF.....QUALIFIED IN MAIN
         (ALPHA .LT. 25.5)
```

Purpose

Activates and deactivates logging of the Debug Session commands and responses.

Format

LOG [OFF]

OFF discontinues writing information to the file. If you subsequently enter another LOG command during the same session, the file is extended and subsequent debug information is written to the file.

Example

```
LOG
```

writes all entered Debug commands and messages to the Debug log file, which is called DEBUGLOG unless you change the name with a DOS or OS/2 SET command.

```
LOG OFF
```

stops writing all Debug commands and messages.

Remarks

All commands and subsequent responses are written to the Debug log file.

Note: The Debug log file is scratched (emptied) before the first log record of a session is written.

The log file is closed when the session is completed.

You can use DOS or OS/2 commands to redirect logging to other files or devices, or to list the contents of a file after a Debug session. See "Changing Devices" on page 7-5.

MACHINE Command: M

Purpose
Displays executable statements and the corresponding object code.

Format

MACHINE [*statement reference*]

Statement reference can be any reference technique (line, label, or offset).

Remarks

When you do not use options with the MACHINE command, the next statement to be executed is displayed.

If a carriage return is entered after a MACHINE Command, the next executable statement is displayed. If a carriage return is entered after an attention interrupt of a MACHINE command, the executable statement that follows the last line displayed is displayed with its object code.

If more lines are to be displayed than can fit on the screen, a message:

- - M O R E - -

is displayed; the display then pauses until the Enter key is pressed.

Each line displayed is preceded by its line number.

The disassembled code will appear in the same format used by DOS or OS/2 DEBUG. That is, memory references will print as hexadecimal numbers and not as symbolic names.

Note that if the source file(s) has been changed, deleted, or moved since the file was compiled, the source line displayed might not correspond to the line requested.

Source lines longer than 73 characters are truncated to 73 characters when displayed.

Example

```
MACHINE 30
```

displays the statement at line 30 and its object code.

```
MACHINE SUBA.410
```

displays the statement at line 410 in subprogram SUBA and its object code.

NEXT Command: N

Purpose

Advances the program by one statement in the executing program unit, then stops as though it had encountered a breakpoint.

Format

NEXT

Remarks

The NEXT command is similar to the STEP command except that the NEXT statement performs subprogram units without stepping through each subprogram statement (unless a breakpoint has been set there).

After entering the NEXT command, you can continue stepping through the program by pressing the Enter key, which is interpreted as another NEXT Command. When you enter a different Debug command, the stepping process is deactivated.

Debug displays the following message at each statement you step through:

NEXT: *line number*[*/label*] IN *program unit*

The *line number* is the line number of the next FORTRAN statement to be executed. The label, */label*, displays when applicable.

The *program unit* is the program unit containing the FORTRAN statement.

Examples

```
NEXT: 2Ø IN MAIN
```

shows that line number 20 will be executed next in program unit MAIN.

```
NEXT: 3Ø/8 IN MAIN
```

shows that line number 30 (having label 8) will be executed next in program unit MAIN.

The statement where the NEXT command halts may contain a breakpoint or WHEN condition. If so, you receive the appropriate message.

QUALIFY Command: Q

Purpose

Changes your qualified program unit to a specified program unit. The qualified program unit determines the meaning of line numbers, statement labels, and unqualified variable names used in subsequent Debug commands. The QUALIFY command does not change the executing program unit.

Format

QUALIFY [*program unit*]

program unit is a name defined within the program by a PROGRAM, SUBROUTINE, or FUNCTION statement. For unnamed main program units, use the special name MAIN.

Remarks

You can only qualify a program unit that has been compiled with the /T compiler option.

The last named qualified program unit remains in effect until you enter another QUALIFY command or a GO, NEXT, RUN, or STEP command.

You can qualify individual variable names and statements without using the QUALIFY command. To do this, preface the variable with the program unit name and separate the two items with a period(.). See 7-9.

If no QUALIFY is used, the executing program unit is always the qualified program unit.

To identify the currently qualified program unit, enter the QUALIFY command without a program unit name. To identify the currently executing program unit use the "WHERE Command: W" on page 7-56.

Examples

If your qualified program unit is MAIN, you can make SUBA your qualified program unit by entering:

```
Q SUBA
```

Debug displays the following message:

```
QUALIFY: SUBA CURRENTLY QUALIFIED
```

If you previously named SUBA as your qualified program unit and you enter the following QUALIFY command:

```
QUALIFY
```

the following message is displayed:

```
QUALIFY: SUBA CURRENTLY QUALIFIED
```

REGISTERS Command: RE

Purpose

Displays the machine registers and flags.

Format

REGISTERS

Example

```
REGISTERS
```

Displays the registers and flags.

Remarks

The registers are displayed in the following format:

```
AX=xxxx  BX=xxxx  CX=xxxx  DX=xxxx  SP=xxxx  BP=xxxx  SI=xxxx  DI=xxxx
DS=xxxx  ES=xxxx  SS=xxxx  CS=xxxx  IP=xxxx   oo dd ii ss zz aa pp cc
```

where xxxx is a hexadecimal register value and:

oo Is the overflow flag (OV set, NV clear)

dd Is the direction flag (DN decrement, UP increment)

ii Is the interrupt flag (EI enable, DI disable)

ss Is the sign flag (NG negative, PL positive)

zz Is the zero flag (ZR zero, NZ not zero)

aa Is the auxiliary carry flag (AC carry, NA no carry)

pp Is the parity flag (PE even, PO odd)

cc Is the carry flag (CY carry, NC no carry)

The math coprocessor registers are not displayed since they are always empty between statements in unoptimized code.

Purpose

Resumes execution from the current breakpoint to the end of the program unit without Debug control.

Format

RUN

Remarks

Execution stops at the first STOP statement (or END, or RETURN statement, which causes a stop). When the program stops, Debug regains control. You can enter Debug commands to inspect the final values of program variables.

Breakpoints, WHEN conditions, and tracing are ignored (not deactivated). When Debug regains control after the RUN statement, these conditions are in effect once again.

You can use attention interrupts to stop the running program and reestablish Debug control.

SET Command

Purpose

Assigns a value to a scalar variable or an array element, or assigns a set of values to an entire array or part of an array.

Format

SET [*set variable*|*array name*] = *value*[,*value*]...

set variable	Is any variable or array element name within your executing program unit. The name cannot be explicitly qualified (preceded by a program unit name).
array name	Is the unqualified name of an array, without subscripts.
value	Is a constant, variable, or array element, optionally preceded by a replication count and/or a sign. The value can be explicitly qualified.

Example

```
SET A = 2
```

sets the variable A to 2.

```
SET M(1) = A
```

sets the array element M(1) to the value A.

Remarks

Operations other than unary negation are not allowed in the value assignments. Mixed mode assignments are allowed for arithmetic data types.

Examples

```
SET A = SUBA.B
```

sets A (in your executing program unit) to the value of variable B. B is in subprogram unit SUBA.

Values are assigned to an array in storage sequence by starting at the beginning and varying the leftmost subscripts most rapidly.

For example, consider I to be a dimensioned array I(2,2). The following SET command:

```
SET I = 1,2,3,4
```

sets the values in the array as follows:

```
I (1,1) = 1
I (2,1) = 2
I (1,2) = 3
I (2,2) = 4
```

You do not have to set values for the entire array. If the value list is exhausted before the end of the array, the remainder of the array remains unchanged.

Multiple assignments can be made by preceding a value with a repeat count and an asterisk (*):

*n**[*value*]

The number, *n*, is an integer value indicating the number of times the value is to be used.

Examples

```
SET A = 2*Ø,3*1.5
```

sets the first two elements to zero, and the next three to 1.5.

To leave a set of sequential array elements unchanged, use a repeat count without a value:

```
SET A = 1.Ø,2*,3.Ø
```

sets the first element to 1.0, leaves the next two elements unchanged, and sets the fourth element to 3.0.

You can use dummy arguments in the SET command as long as they exist in the executing program unit. By doing so, you might change a variable in the calling program unit.

SOURCE Command: SO

Purpose

Displays lines of the source file (or files) that correspond to the designated qualified program unit.

Format

SOURCE [*statement reference*]
SOURCE *statement range*
SOURCE *program unit*

statement reference	Can be by line, label, or offset. The referenced line may be any line in the qualified program unit. If the referenced line is part of a statement that has continuation lines, all lines of the statement are displayed.
statement range	Is two statement references separated by a colon. The statements and comments within the specified range are displayed. If the second statement reference is not explicitly qualified, the qualified subprogram unit for the first statement reference is used. The two statement references must refer to statements or comments in the same program unit.
program unit	Is a name defined within the program by a PROGRAM, SUBROUTINE, or FUNCTION statement. For unnamed main program units, use the special name MAIN. The first line of the specified program unit is displayed.

Example

```
SOURCE 10:15
```

displays all statements and comments from line 10 to line 15 in the qualified program unit.

```
SOURCE SUBA.121:126
```

displays all statements and comments from line 121 to line 126 in the subprogram SUBA.

```
SOURCE /30
```

displays the statement at the line labeled 30. If 30 is the line number of a statement that is continued, all of the lines of the designated statement are displayed.

Remarks

When you do not use options with the SOURCE command, the next statement to be executed is displayed.

If a carriage return is entered after a SOURCE Command, the next statement or command (if any) in the same program unit will be displayed. If a carriage return is entered after an attention interrupt of a SOURCE command, the statement or comment designated by the next line that would have been displayed, is displayed.

If more lines are to be displayed than can fit on the screen, a message:

- - M O R E - -

is displayed; the display then pauses until the Enter key is pressed.

Each line displayed is preceded by its line number.

If the source file(s) has been changed, deleted, or moved since the file was compiled, the source lines displayed might not correspond to the lines requested.

Block data source lines cannot be displayed by the SOURCE command.

SOURCE Command: SO

Source lines longer than 73 characters are truncated to 73 characters when displayed.

If the specified range in the SOURCE command includes lines not in the currently qualified (or explicitly qualified) program unit, those lines are not displayed. An error occurs if no lines are in the range for the program unit.

You may easily display the current program unit without having to know the exact line numbers. For example:

```
SOURCE   1:65535
```

will display the entire current qualified program unit.

```
SOURCE  SUB.1:65535
```

will display the entire program unit named SUB.

This does not allow you to display lines outside the currently qualified program (or explicitly qualified program unit). If lines for multiple program units are to be displayed, a separate SOURCE command must be entered for each program unit with the appropriate qualification.

Purpose

Advances the program by one statement, then stops as though it had encountered a breakpoint.

Format

STEP

Remarks

The STEP command is similar to the NEXT command, except that it steps through each statement of called subprograms.

Once you enter a STEP command, you can continue stepping through your program by pressing the Enter key, which is interpreted as another STEP Command.

You can stop stepping through your program by entering another Debug command.

Debug displays the following message at each statement you step through:

STEP: *line number*[*/label*] IN *program unit*

The *line number* is the line number of the next FORTRAN statement to be executed. The label, */label*, displays when applicable.

The *program unit* is the program unit containing the FORTRAN statement.

Examples

```
STEP: 125 IN MAIN
```

This message shows that line number 125 will be executed next in program unit MAIN.

TRACE Command: T

Purpose

Activates and deactivates statement tracing in part or all of your qualified program unit.

Format

TRACE [*statement range*|OFF]

statement range	Is two statement references separated by a colon. The statements within the specified range are traced. If the second statement reference of the statement range is not explicitly qualified, the qualification used for the first statement reference will be applied to the second.
OFF	Deactivates the trace.

Example

```
TRACE 10:15
```

traces all statements from line 10 to line 15.

```
TRACE 10:/20
```

traces all statements from line 10 to statement label 20.

Remarks

When you do not use options with the TRACE command, every statement in your qualified program unit is traced.

Trace can only be active for one statement range or one program unit at a time.

Debug displays the following message each time a statement is executed:

TRACE: *line number*[/*label*] IN *program unit*

The *line number* is the line number of the FORTRAN statement which is to be executed next. The label, */label*, is displayed when applicable.

The *program unit* is the program unit containing the FORTRAN statement.

An example of a trace message series is:

```
TRACE: 10 IN MAIN
TRACE: 11/4 IN MAIN
TRACE: 12 IN MAIN
TRACE: 13 IN MAIN
TRACE: 14 IN MAIN
TRACE: 15/5 IN MAIN
```

WHEN Command: WN

Purpose

Defines conditions under which breakpoints occur. This command is used to:

- Monitor the change in the value of a variable
- Test the value of a logical variable
- Test a relational expression.

Format

WHEN *condition number* [@*variable*]
WHEN *condition number* [([.NOT.] *variable*)]
WHEN *condition number* [(*rel-expression*)]
WHEN *condition number* OFF
WHEN *[OFF]

condition number	Is a one-digit number (1 through 9) that identifies a specific WHEN condition.
variable	Is the variable whose value is being monitored.
rel-expression	Is a relational expression.
asterisk (*)	stands for all defined conditions.

Example

```
WHEN 1 @TOTAL
```

sets a breakpoint whenever the value of TOTAL in the qualified program unit changes. This condition is identified by the number 1.

```
WHEN 2 @SUBA.TOTAL1
```

sets a breakpoint whenever the value of TOTAL1 in subprogram unit SUBA changes. This condition is identified by the number 2.

The initial value of the variable is established at the entry of the WHEN command.

Remarks

Conditions specified in the WHEN command are monitored or tested within all program units at the start of each executable statement.

When monitoring a variable for a change in value, storing the same value will not trigger the WHEN condition.

As soon as a WHEN condition is met, Debug suspends program execution as though it has encountered a breakpoint. However, the occurrence of a WHEN condition does not end the monitoring of that condition. This means that if you reach a WHEN condition breakpoint and then continue executing with the specific condition unchanged, another breakpoint occurs at the next statement. This continues until the status of the WHEN condition changes or until you deactivate the condition.

The use of dummy variables or variably subscripted array elements is not allowed in the WHEN Command.

To test whether a logical variable is true or not true, enter the WHEN command in one of the following formats:

WHEN *condition number* (*variable*)
WHEN *condition number* (.NOT. *variable*)

Note: There must be at least one space between the .NOT. operator and the *variable*.

Examples

```
WHEN 3 (LOGVAR)
WHEN 4 (.NOT. SUBB.LOGVARI)
```

The first WHEN command (with a condition number of 3) sets a breakpoint whenever the value of LOGVAR in the qualified program unit is true. The second WHEN command sets a breakpoint whenever the value of LOGVARI in subprogram unit SUBB is false.

To test the relational value of a variable, enter the WHEN command in the following format:

WHEN *condition number* (*rel-expression*)

WHEN Command: WN

The breakpoint occurs when the relational expression is true. Mixed mode is allowed for arithmetic variables.

The relational operators are:

Operator	Mnemonic	Symbol
Equal	.EQ.	=
Not equal	.NE.	<> or ><
Greater than	.GT.	>
Greater than or equal to	.GE.	>= or =>
Less than	.LT.	<
Less than or equal to	.LE.	<= or =<

The mnemonic relational operators must be preceded and followed by at least one space. Either the mnemonic or symbol is valid in the WHEN command. For complex and character data, use only the equal or not equal operators.

Examples

```
WHEN 5 (ALPHA .LT. 25.5)
```

This command sets a breakpoint when the value of the variable ALPHA in the qualified program unit is less than 25.5.

```
WHEN 6 (BETA .EQ. ZETA)
```

This command sets a breakpoint whenever the character variable BETA in the qualified program unit is equal to ZETA.

```
WHEN 7 (SUBC.GAMMA >= 44.01)
```

This command sets a breakpoint when the value of the variable GAMMA, in the subprogram SUBC, is greater than or equal to 44.01.

To deactivate a specified WHEN condition, enter:

WHEN *condition number* OFF

The condition is deactivated but not canceled. To reactivate the same condition, enter:

```
WHEN condition number
```

To change the WHEN condition, just enter a new WHEN Command using the same condition number. Deactivation is not required.

To deactivate all defined WHEN conditions, enter:

```
WHEN * OFF
```

To reactivate all defined WHEN conditions, enter:

```
WHEN *
```

When a WHEN condition is met, the following message is displayed:

```
WHEN: condition number QUALIFIED IN
   program unit condition OCCURRED AT
   line number[/label] IN
   program unit
```

The first *program unit* is the program unit qualified at the time the WHEN condition was defined. The second is the program unit in which the WHEN condition was triggered.

The *line number* is the line number where the WHEN condition occurred. The label,*/label*, displays when applicable.

The *condition* is the condition that triggered the breakpoint.

Examples

```
WHEN: 5 QUALIFIED IN MAIN (ALPHA .LT. 25.5)
          OCCURRED AT 183/66 IN MAIN
```

This is a displayed message. It shows that WHEN condition number 5 was triggered at line number 183 (which also has the label 66) of program unit MAIN. The condition occurred because the specified relational expression, (ALPHA .LT. 25.5), became true.

WHERE Command: W

Purpose

Displays the location of your current breakpoint, and optionally tracks back through your program's calling sequence.

Format

WHERE [TRACK|TK]

TRACK	Requests a listing of the chain of subprogram calls leading to the current program unit.
TK	Abbreviation for TRACK

Remarks

When the trackback option, TRACK, is not used, the following message is displayed:

WHERE: *line number[/label]* IN *program unit*

The *line number* is the line number of the current breakpoint. The label, */label*, displays when applicable.

The *program unit* is the qualified program unit containing the breakpoint.

Examples

```
WHERE: 77 IN SUBB
WHERE: 79/40 IN SUBB
```

When the trackback option, TRACK, is used, the first message displayed is your current position followed by one or more messages:

prog-unit-1 CALLED BY *prog-unit-2* AT *line number[/label]*:

The *prog-unit-1* is the program unit that was entered. The *prog-unit-2* is the program unit that issued the call.

The *line number* is the line number of the call. The label, */label*, is displayed when applicable.

Examples

```
WHERE: 77 IN SUBB
             SUBB CALLED BY SUBA AT 10/25
             SUBA CALLED BY MAIN AT 125
```

When you enter WHERE as the first Debug command of a Debug session, the following message is displayed:

INITIAL ENTRY IN *programname*

where *programname* is the name of the main program. If the main program does not have a name, the name MAIN is displayed.

Exercises Using Debug

The rest of this chapter presents four basic exercises using the facilities of the Interactive Symbolic Debug. Using these exercises, you will be guided through:

- Setting breakpoints
- Listing and setting variables
- Tracing program execution
- Displaying source lines, corresponding object code, and machine registers

Preparing the Demonstration Program for Debug

To compile the demonstration program under Debug control, and redirect output to your printer (if configured), enter the following Compile Command:

```
FORTRAN DEMO /TL > PRN
```

or if you do *not* have a math coprocessor:

```
FORTRAN DEMO /TNL >PRN
```

All program and subprogram units in the file DEMO.FOR are now compiled under Debug control, and can be used in the exercises that appear throughout this section.

To link the demonstration program for execution under Debug control, enter the following LINK Command:

```
LINK DEMO;
```

making sure the appropriate IBM FORTRAN/2 library is available. See "Linking the Sample Program" on page 2-25.

Setting Breakpoints in DEMO

Use the demonstration program to set breakpoints, establish WHEN conditions, and display the results on your screen, according to the following instructions.

1. Enter DEMO in response to the DOS or OS/2 Command promp You should see the following message:

   ```
   IBM FORTRAN/2 Interactive Symbolic Debug
   Version 1.00
   (C)Copyright IBM Corp 1984, 1987
   (C)Copyright Ryan-McFarland Corp 1984, 1987

   INITIAL ENTRY IN DEMO
   -=>
   ```

   Your Debug Session is now started.

2. Set breakpoints at line 17 and at label 10.

   ```
   AT (17,/10)
   ```

3. Set breakpoints at line 23 of DEMO, and at an offset of one lin from label 10 in subprogram unit MYSIN.

   ```
   AT (23,MYSIN./10-1)
   ```

4. Set breakpoints at the entry point and the exit point of MYSIN.

   ```
   AT MYSIN
   ```

5. Set a WHEN condition breakpoint in DEMO.

   ```
   WHEN 2 (X(1,1) .LT. 0.5)
   ```

6. List the breakpoints set.

   ```
   LISTBRKS
   ```

7. You will see the following:

   ```
   CURRENT BREAKPOINTS
   ..... 17 IN DEMO
   ..... 21/10 IN DEMO
   ..... 23 IN DEMO
   .....(ENTRY/EXIT) IN MYSIN
   ..... 39 IN MYSIN
   CURRENT WHEN CONDITIONS
     2  ON.... QUALIFIED IN DEMO
                  (X(1,1) .LT. 0.5)
   ```

8. Now, deactivate these breakpoints.

```
AT * OFF
AT MYSIN./1Ø-1 OFF
AT MYSIN OFF
WHEN 2 OFF
```

9. List the breakpoints again.

```
LISTBRKS
```

You will see:

```
CURRENT BREAKPOINTS
   NO BREAKPOINTS ARE SET
CURRENT WHEN CONDITIONS
   2  OFF... QUALIFIED IN DEMO
                (X(1,1) .LT. Ø.5)
```

10. Now, use the QUALIFY Command to change your Qualified Program Unit from DEMO to FACTRL:

```
QUALIFY FACTRL
```

You will see the following message:

```
QUALIFY: FACTRL CURRENTLY QUALIFIED
```

11. Set a breakpoint in FACTRL, and then direct that DEMO again become your Qualified Program Unit:

```
AT /1Ø
QUALIFY DEMO
```

You will see:

```
QUALIFY: DEMO CURRENTLY QUALIFIED
```

12. List the breakpoints.

```
LISTBRKS
```

You will see:

```
CURRENT BREAKPOINTS
..... 49/1Ø IN FACTRL
CURRENT WHEN CONDITIONS
  2  OFF... QUALIFIED IN DEMO
               (X (1,1) .LT. Ø.5)
```

13. Now, remove the breakpoint set in FACTRL.

```
AT FACTRL./10 OFF
```

14. The END Command is described in "END Command" on page 7-23. However, use it now to end your Debug Session. Enter:

```
END
```

and your Debug Session is over. The following message will appear on your screen:

```
*** - = >  THE DEBUG RUN HAS FINISHED  < = - ***

Execution terminated  :  0
```

Listing and Setting Variables in DEMO

In this section, you will use the demonstration program to list variables and set values for variables. You will also use commands to resume execution.

1. Enter DEMO in response to the DOS or OS/2 Command prompt. You will see the following message:

```
IBM FORTRAN/2 Interactive Symbolic Debug
Version 1.00
(C)Copyright IBM Corp 1984, 1987
(C)Copyright Ryan-McFarland Corp 1984, 1987

INITIAL ENTRY IN DEMO
-=>
```

2. Set the values of arrays X and NUMBER to zero.

```
SET X = 30*0
SET NUMBER = 10*0
```

3. Check the results.

```
LIST (X,NUMBER)
```

You will see:

```
X =
   0.00000000E+00  0.00000000E+00  0.00000000E+00  0.00000000E+00
   0.00000000E+00  0.00000000E+00  0.00000000E+00  0.00000000E+00
   0.00000000E+00  0.00000000E+00  0.00000000E+00  0.00000000E+00
   0.00000000E+00  0.00000000E+00  0.00000000E+00  0.00000000E+00
   0.00000000E+00  0.00000000E+00  0.00000000E+00  0.00000000E+00
   0.00000000E+00  0.00000000E+00  0.00000000E+00  0.00000000E+00
   0.00000000E+00  0.00000000E+00  0.00000000E+00  0.00000000E+00
   0.00000000E+00  0.00000000E+00
NUMBER =
             0               0               0               0
             0               0               0               0
             0               0
```

4. Start the first loop of the DO-loop in DEMO, and stop before executing the statement at line 18. To do this, first enter a breakpoint at line 18.

```
AT 18
```

and then enter:

```
GO
```

You will see:

```
             X          MYSIN(X)        SIN(X)          TERMS
AT: 18 IN DEMO
```

5. Check to see how many times you have been around the DO-loop by examining the values of I, VAL and X(I,1).

```
LIST (I, VAL, X(I,1))
```

You will see:

```
I = 1
VAL = 0.10000000E+00
X(1,1) = 0.00000000E+00
```

6. Go around the loop once more:

```
GO
```

You'll once again see:

```
AT: 18 IN DEMO
```

7. Now list the variables again (repeat Step 5). This time you will see:

```
I = 2
VAL = 0.20000000E+00
X(2,1) = 0.00000000E+00
```

By setting a single breakpoint in a DO-loop, you have stepped through it, one loop at a time.

8. Now, step through your program. Enter:

```
NEXT
```

You will see:

```
NEXT: 19 IN DEMO
```

9. Enter NEXT again. You will see:

```
NEXT: 20 IN DEMO
```

10. Press the ENTER Key. You will see:

```
NEXT: 21/10 IN DEMO
```

11. Press the ENTER Key again. You will see:

```
NEXT: 17 IN DEMO
```

12. Press the ENTER Key one more time. You will see:

```
NEXT: 18 IN DEMO
AT: 18 IN DEMO
```

which means that line 18 was both the next statement and had a breakpoint set.

13. List the variables:

```
LIST (I, VAL, X(I,1))
```

You will see:

```
I = 3
VAL = 0.30000001E+00
X(3,1) = 0.00000000E+00
```

14. Using the NEXT Command above executed (without stepping through) the subprogram unit MYSIN. This time, step through MYSIN. Enter:

```
STEP
```

You will see the following:

```
STEP: 19 IN DEMO
```

15. Press the ENTER Key four times. The following messages will be displayed, one at a time:

```
STEP: 36 IN MYSIN
STEP: 37 IN MYSIN
STEP: 38 IN MYSIN
STEP: 47 IN FACTRL
```

You have stepped through MYSIN and are now in the FUNCTION subprogram FACTRL (and it is now your Qualified Program Unit).

16. List the variable I in FACTRL.

```
LIST I
```

You will see:

```
I = 3
```

17. Continue executing your program until it reaches a breakpoint. Enter:

```
GO
```

You will see:

```
AT: 18 IN DEMO
```

18. Execute your program until it reaches a STOP Statement. Enter:

```
RUN
```

You will see:

```
    Ø.1         Ø.Ø9983333         Ø.Ø9983342        2
    Ø.2         Ø.19866934         Ø.19866933        3
    Ø.3         Ø.29552Ø25         Ø.29552Ø22        3
    Ø.4         Ø.38941833         Ø.38941833        4
    Ø.5         Ø.47942552         Ø.47942555        4
    Ø.6         Ø.56464249         Ø.56464249        4
    Ø.7         Ø.64421773         Ø.64421767        5
    Ø.8         Ø.717356Ø9         Ø.717356Ø9        5
    Ø.9         Ø.78332698         Ø.78332692        5
    1.Ø         Ø.84147Ø96         Ø.84147Ø96        5
STOP STATEMENT AT 23 IN DEMO
```

19. DEMO is once again your Qualified Program Unit. List the variables in DEMO.

```
LIST (I, VAL, X(I,1))
```

This time, you will see:

```
I = 4
VAL = Ø.1ØØØØØØØE+Ø1
X(4,1) = Ø.4ØØØØØØ1E+ØØ
```

20. If you enter any of the resume execution commands from a STOP Statement, you are returned to the operating system. However, you can enter a GO Command with a line number to continue debugging:

```
GO 14
```

You will see:

```
    X                MYSIN(X)        SIN(X)          TERMS
AT: 18 IN DEMO
```

You just branched from the STOP Statement at line 23, to the PRINT Statement at line 14, and then resumed execution until the breakpoint at line 18 was reached.

21. Experiment with these commands on your own or conclude your Debug Session. To do the latter, enter:

```
END
```

The following message will appear on the screen:

```
*** - = > THE DEBUG RUN HAS FINISHED < = - ***

Execution terminated  :  Ø
```

Tracing DEMO

In this exercise, you will use the TRACE and ENTRY Commands to trace execution through a range of statements, and through the entry and exit points of DEMO.

1. Execute your program as you have done before. In response to the Debug command prompt, enter the following commands (one at a time):

```
AT 21
TRACE
GO
```

You will see the following:

```
TRACE: 14 IN DEMO
        X                MYSIN(X)         SIN(X)       TERMS
TRACE: 16 IN DEMO
TRACE: 17 IN DEMO
TRACE: 18 IN DEMO
TRACE: 19 IN DEMO
TRACE: 20 IN DEMO
AT: 21/10 IN DEMO
```

You have now traced your program from the beginning of the breakpoint through the first loop of the DO-loop.

2. Now use the TRACE Command with a specified range of statements. Enter the following (one at a time):

```
TRACE OFF
TRACE 16:18
GO
```

You will see:

```
TRACE: 17 IN DEMO
TRACE: 18 IN DEMO
AT: 21/10 IN DEMO
```

Note that in this message, line 16 is not listed. This is because the DO-loop being traced does not include the initial DO Statement.

3. Now, to stop the message output, set a breakpoint in MYSIN and deactivate TRACE. Enter (one at a time):

```
AT MYSIN.39
TRACE OFF
```

4. Reactivate tracing, and activate subprogram entry and exit tracing as well. Enter:

```
TRACE 18:20
ENTRY
GO
```

You will see:

```
TRACE: 18 IN DEMO
TRACE: 19 IN DEMO
ENTRY: MYSIN ENTERED AT MYSIN
ENTRY: FACTRL ENTERED AT FACTRL
ENTRY: RETURN FROM FACTRL
AT: 39 IN MYSIN
```

5. To set the stage for dummy argument listing, return to DEMO.

```
AT MYSIN.39 OFF
ENTRY OFF
GO
```

You will see:

```
TRACE: 20 IN DEMO
AT: 21/10 IN DEMO
```

6. Now enter:

```
AT MYSIN.36
ENTRY PARM
GO
```

You will see:

```
TRACE: 18 IN DEMO
TRACE: 19 IN DEMO
ENTRY: MYSIN ENTERED AT MYSIN
P1 = Value 1
P2 = Value 2
AT: 36 IN MYSIN
```

Note that for this display, the values for P1 and P2 are left unspecified in this guide. That is because those values depend on the number of times you have gone through the DO-loop. Use the LIST Command to see these values:

```
LIST (P1, DEMO.VAL, P2, DEMO.N)
```

You will see:

```
P1 = Value 1
VAL = Value 1
P2 = Value 2
N = Value 2
```

7. To conclude your Debug Session. Enter:

```
END
```

The following message will appear on the screen:

```
*** - = > THE DEBUG RUN HAS FINISHED < = - ***

Execution terminated : 0
```

Examining Source, Object and Register Contents

In this exercise, you'll use commands that allow you to look at both source and object code, and examine the contents of registers.

1. Enter DEMO in response to the DOS or OS/2 command prompt.

   You will see the following:

   ```
   IBM FORTRAN/2 Interactive Symbolic Debug
   Version 1.00
   (C)Copyright IBM Corp 1984, 1987
   (C)Copyright Ryan-McFarland Corp 1984, 1987

   INITIAL ENTRY IN DEMO
   -=>
   ```

2. List the source code in DEMO from line 11 to line label 10.

   ```
   SOURCE 11:/10
   ```

   You will see:

   ```
   11      PROGRAM DEMO
   12      REAL MYSIN,X(10,3)
   13      INTEGER NUMBER(10)
   14      PRINT  '(10X,''X'',10X,''MYSIN(X)'',11X,''SIN(X)'',11X
   15    *        ''TERMS'',/)'
   16      DO 10 I = 1, 10
   17      VAL = I * 0.1
   18      X(I,1) = VAL
   19      X(I,2) = MYSIN(VAL,N)
   20      X(I,3) = SIN(VAL)
   21 10   NUMBER(I) = N
   -=>
   ```

3. Now, set a breakpoint at line 19 and then start execution.

   ```
   AT 19
   GO
   ```

   You will see:

   ```
     X          MYSIN                SIN(X)              TERMS

   AT: 19 IN DEMO
   -=>
   ```

4. Examine the object code generated for line 19.

```
MACHINE 19
```

You will see something similar to the following:

```
    19                    X(I,2) = MYSIN(VAL,N)
ssss:00A0   BE1000             MOV    SI,0010
ssss:00A3   9A28004D12         CALL   1240:0029
ssss:00A8   BA491B             MOV    DX,1B49
ssss:00AB   8EDA               MOV    DS,DX
ssss:00AD   8B361400           MOV    SI,[0014]
ssss:00B1   D1E6               SHL    SI,1
ssss:00B3   D1E6               SHL    SI,1
ssss:00B5   1E                 PUSH   DS
ssss:00B6   B81000             MOV    AX,0010
ssss:00B9   50                 PUSH   AX
ssss:00BA   1E                 PUSH   DS
ssss:00BB   B91800             MOV    CX,0018
ssss:00BE   51                 PUSH   CX
ssss:00BF   8976F4             MOV    [BP-0C],SI
ssss:00C2   33F6               XOR    SI,SI
ssss:00C4   9A16002012         CALL   1220:0016
ssss:00C9   BB491B             MOV    BX,1B49
ssss:00CC   8EDB               MOV    DS,BX
ssss:00CE   8B76F4             MOV    SI,[BP-0C]
ssss:00D1   9B                 WAIT
ssss:00D2   D99C4400           FSTP   SHORT REAL[SI+0044]
```

You may see different hex values displayed than shown above depending upon where the program was loaded. ssss is the segment selector for the code.

5. Use the REGISTERS command to examine the value stored in the CS register:

```
REGISTERS
```

You will see something similar to the following:

```
AX=xxxx  BX=xxxx  CX=xxxx DX=xxxx  SP=xxxx  BP=xxxx  SI=xxxx  DI=xxxx
DS=xxxx  ES=xxxx  SS=xxxx CS=ssss  IP=00A0   NV UP EI PL NZ NA PO NC
```

xxxx is the hex value displayed in the registers.

6. Look at lines ssss:00B4 and ssss:00B7 of the object examined in step 4.

```
ssss:ØØA8   BAxxxx       MOV    DX,xxxx
ssss:ØØAB   8EDA         MOV    DS,DX
```

These lines set up the DS register to point to the local data segment. Use the address these lines are moving into the DS register to display the value of the variable I.

The next line is:

```
ssss:ØØAD   8B3614ØØ     MOV    SI,[ØØ14]
```

This line moves the offset of the variable I into the SI register. You can use this information to examine the contents of memory where I is located. To do this enter:

```
DISPLAY xxxx:ØØ14
```

xxxx is the segment selector that lines ssss:00A8 and ssss:00AB moved into the DS register.

You will see:

```
xxxx:0010      01 00-00 00 00 00 00 00 00 00 00 . . . . . . . .
```

7. To end the Debug session. Enter:

```
END
```

You will see:

```
*** - = > THE DEBUG RUN HAS FINISHED < = - ***

Execution terminated  :  Ø
```

Chapter 8. Interfaces with Other IBM Languages and Products

To communicate with the the operating system or ROM BIOS, it may be useful to call an assembler language subprogram assembled with the Macro Assembler/2 Similarly, other hardware interfaces, such as the IBM Math Co-Processor, may be accessed in this way. For example, an assembler language subprogram can be used to access the Asynchronous Communications Adapter or communicate with a special adapter card.

This chapter explains the interface requirements between programs written in IBM FORTRAN/2 and assembler language subprograms. It assumes that you are familiar with assembler language and the Macro Assembler/2 For more information, see the "EXTERNAL Specification Statement" and the "CALL Control Statement" in the *IBM FORTRAN/2 Language Reference* manual before reading this section.

This chapter also explains how to call OS and Application Programming Interface functions and describes the interface requirements.

Structuring your Assembler Language Subprogram

Your assembler language subprogram typically contains three segment definitions (SEGMENT and ENDS statements enclosing other statements.) These definitions allow you to define your program code (CODE segment), local data and trackback information (DATA segment), and stack space (STACK segment). You may define additional segments for your own special needs. As the following sample program structure shows, you may also use the STRUC feature of the Macro Assembler/2 to lay out your parameter block that is passed on the stack.

Example

```
          TITLE     MYSUB
          PAGE      ,132

PARMBLK   STRUC
          DW        ?         ; Old BP
          DD        ?         ; Return address
; You can put a definition of your parameters
; here.
VARB      DD        ?
VARA      DD        ?
PARMBLK   ENDS
; The following four entries are for
; trackback. See "DATA Segment" on page 8-4
; for more information.
LAA@MYSUB SEGMENT   'F@DATA'
          DD        MYSUB
          DD        Ø
          DB        5         ; Number of characters in MYSUB
          DB        'MYSUB'
; Local data areas and values needed
; for your assembler language subprogram

          •
          •
          •
LAA@MYSUB ENDS

P@MYSUB   SEGMENT   'CODE'
          ASSUME CS:P@MYSUB,DS:LAA@MYSUB
; Your CODE segment
; must contain a "far proc"
```

```
MYSUB     PROC      FAR
PUBLIC    MYSUB
;
; Save caller's BP
;
          PUSH      BP
;
; Set BP to point to parameter block
;
          MOV       BP,SP
;
; Save information for trackback.
; This is only needed if
; procedure calls IBM FORTRAN/2
;
          MOV       AX,OFFSET LAA@MYDATA
          PUSH      AX
          PUSH      DS
; Setup DS to point to
; procedure's DATA segment.
;
          MOV       AX,LAA@MYDATA
          MOV       DS,AX
; Your assembler logic goes here
; Restore SP and BP and
; return to caller.
;
          MOV       SP,BP
          POP       BP
          RET       8    ;Pop parameters and return address
MYSUB     ENDP
P@MYSUB    ENDS
STACK     SEGMENT   WORD STACK 'STACK'
          DB        stack_size DUP(?)
STACK     ENDS
          END
```

The three segments in your assembler language subprogram each have special requirements. Those requirements are listed below.

CODE Segment

The CODE segment is the part of your assembler language subprogram that contains the assembler language instructions that make up the logic flow. There are several special requirements for this CODE segment:

1. It should have a CLASS of 'CODE'.
2. It must contain the entry point as the name of a far procedure (PROC FAR).
3. You must identify the entry point to your subprogram by including a PUBLIC definition.
4. See "Rules for Coding Your Assembler Language Subprogram" on page 8-7 for more details.

DATA Segment

Your assembler language subprogram must contain a DATA segment that defines data values, areas needed for your program, and trackback information. There are several special requirements for the DATA segment:

1. It should have a class of 'DATA' or 'F@DATA'.
2. You should set the DS register value to that of your DATA segment. You should also include an ASSUME statement to reflect this.
3. The DATA segment may need to contain trackback information. This information is required only if your assembler language subprogram calls an IBM FORTRAN/2 subprogram (directly or indirectly) and that subprogram has the potential of causing a trackback to occur. This information is:

```
DD      procname
DD      0
DB      Number of characters in procname
DB      'procname'      ; 1 to 31 characters
```

STACK Segment

Every assembler language subprogram should have a STACK segment. You should reserve space in the STACK segment for:

- Subprogram arguments
- Temporary storage of values
- Pushed values (other than arguments to other subprograms)
- Space required to link with the subprogram's caller (10 bytes).

There are several special requirements for the STACK segment:

1. It must be named STACK and have a class of 'STACK'.
2. It should be declared last in your subprogram.
3. The stack should contain only uninitialized data.

Receiving Control From the Caller

When your assembler language subprogram receives control, the following is set up for you:

- The math coprocessor floating point stack is empty.
- Any arguments to the subprogram are pushed onto the stack immediately before the FAR CALL to the subprogram. The instructions at the beginning of the subprogram's CODE segment set up SS:BP to contain the double word address (segment base and offset) of a parameter block which allows the called subprogram to access the actual arguments on the CALL statement.

The first 6 bytes of the parameter block contain information needed to return to the subprogram's caller. Therefore, the argument list actually starts at SS:BP+6.

Arguments are pushed onto the stack left to right. Therefore, their order in the parameter block is the reverse of the order of actual arguments in the CALL statement.

The form of each argument of type CHARACTER*n is different from the forms of arguments of other types. For non-character arguments, the argument is a doubleword address of the actual argument. For

CHARACTER*n, the argument is a word length of the actual argument followed by the doubleword address of the actual argument. Note that alternate return arguments do *not* appear in the parameter block.

If your assembler language subprogram implements a FORTRAN CHARACTER*n function, the first argument in the parameter block is a word length and a doubleword address for the location of where the function result should be placed. See "Returning to the Caller" on page 8-8.

The $VAL intrinsic function can be used to place INTEGER*2 and INTEGER*4 values in the parameter block.

The storage locations pointed to by the addresses in the parameter list should be referenced to obtain the values passed into your assembler language subprogram. If your subprogram returns a value to the caller through a variable in the CALL statement actual argument list, then the address in the parameter block points to the location of the variable. Assign the value to the memory location pointed to by the address specified in the parameter block. You should only assign values to variables. You should never assign a new value to an actual argument that is a constant or expression.

If the actual argument is a subprogram name (procedural parameter), the doubleword address in the parameter block is the entry point address to the subprogram passed.

Rules for Coding Your Assembler Language Subprogram

In your assembler language subprogram, you may freely use any register except SS, BP, and SP. If you change the values of these registers, you must restore them before you return to IBM FORTRAN/2. You may alter the DS register without restoring it.

Before starting the logic of your assembler subprogram, you should:

1. Push BP on the stack and set BP to SP. The old BP is now saved on the stack so it can be restored when returning to the caller. SS:BP is now set up to point to the parameter block.
2. Push the offset of the subprogram's local data area onto the stack. Push DS onto the stack. These actions are required for trackback purposes.
3. Set DS to point to the local data area. Your local data segment can now be referenced.

You can use the stack to store temporary values during the execution of your assembler language subprogram. To accomplish this, you should do the following:

1. On entry, subtract from the SP register the amount of bytes you need to reserve for your temporary values.
2. Before you exit your assembler language subprogram, set the SP register equal to the value in the BP register. This frees the memory area you reserved earlier.

Returning to the Caller

If your assembler language subprogram implements an IBM FORTRAN/2 SUBROUTINE, and you return with an alternative return, you must place the alternative return index in the AX register on exit from your subprogram. This specifies to which actual argument (associated with an asterisk) control is to be returned. An index between 1 and the number of labels, inclusive, indicates an alternate return. Any other value indicates execution should continue with the statement after the CALL.

If your assembler language subprogram is a function, that is, it returns a value to IBM FORTRAN/2 via a function call, then you must provide the function return value on exit from your subprogram. The following describes the method for returning values for different types of functions:

- If the function return value is LOGICAL*n, then the result is placed in AL.
- If the function return value is INTEGER*2, then the result is placed in AX.
- If the function return value is INTEGER*4, then the result is placed in DX/AX. The most significant part of the value is placed in DX; the least significant in AX.
- If the function return value is REAL*4 or REAL*8 (DOUBLE PRECISION), then the result should be placed in the top stack item on the math coprocessor register stack (ST). This value must be the only item on the floating-point stack.

- If the function return value is COMPLEX*n, then the real part is placed in the top stack item on the math coprocessor register stack (ST), and the imaginary part is placed in the next register item (ST(1)). These two items must be the only items on the floating-point stack.
- If the function return value is CHARACTER*n, then the first argument in the parameter block is a word length and a doubleword address of the area where the function result should be placed. You must fill this area with the function result for the length specified in the parameter block.

IMPORTANT: When your subprogram returns, you must ensure that the direction flag is clear. IBM FORTRAN/2 assumes that it is in a clear state.

When your assembler language subprogram returns to the caller, you must make sure that the SS value is the same value that it contained on entry to your subprogram. Also, you must ensure that the stack is in the same state as it was on entry. That is, the SP and BP registers must have the same values as they did on entry.

This can be accomplished by:

```
MOV   SP,BP
POP   BP
```

These instructions remove any temporaries allocated on the stack and the doubleword address of your local data area that was pushed on the stack.

Also, the math coprocessor register stack must be empty when your subprogram returns. The only exception to this is returning values for REAL*n and COMPLEX*n functions as described above.

To return to the caller, use a FAR RET instruction:

```
RET  n
```

where *n* equals 2c + 4p + 6r-2k. The letter *c* is the number of character arguments; *p* is the number of arguments; *r* is 0 for a subroutine or non-character function and 1 for a character function; and *k* is the number of $VAL arguments of INTEGER*2 type.

The RET instruction should remove the parameters from the stack.

Sample Assembler Language Subprogram

The following is an example of an assembler language subprogram:

This sample program contains a IBM FORTRAN/2 program called EXMP1 that calls an assembler language subprogram called FORSUB. The subroutine FORSUB calculates the sum and truncated 8 bits mean of the last three arguments, and puts the mean in the first argument, and the sum in the second argument. All the variables passed to the subprogram are of type INTEGER*2.

MAIN.FOR:

```
      PROGRAM EXMP1
      INTEGER*2 I,J,K,MEAN,ISUM
      I = 3
      J = 4
      K = 5
      CALL FORSUB(MEAN,ISUM,I,J,K)
      PRINT *, MEAN, ISUM, I, J, K
      STOP
      END
```

FORSUB.ASM:

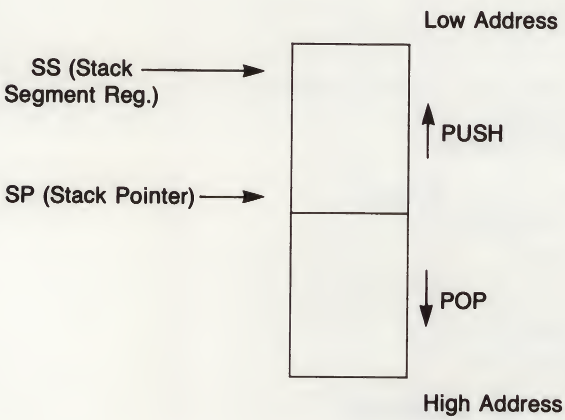
```
         PAGE     ,132
         TITLE    FORTRAN SUBROUTINE
;PARAMETER LIST CONTROL BLOCK
FRAME    STRUC
;OLD BP
         DW       ?
;RETURN ADDRESS
         DD       ?
;ADDR OF WORD INTEGER - THIRD VALUE
PARMVAL3 DD       ?
;ADDR OF WORD INTEGER - SECOND VALUE
PARMVAL2 DD       ?
;ADDR OF WORD INTEGER - FIRST VALUE
PARMVAL1 DD       ?
;ADDR OF WORD TO RECEIVE SUM
PARMSUM  DD       ?
;ADDR OF WORD TO RECEIVE MEAN
PARMEAN  DD       ?
FRAME  ENDS
```

```
LAA@FORSUB  SEGMENT  'F@DATA'
; Trackback information is not needed
; since FORSUB does not call any
; IBM FORTRAN/2 Subprograms
; (directly or indirectly).
LAA@FORSUB  ENDS

P@FORSUB SEGMENT  'CODE'
         ASSUME  CS:P@FORSUB,DS:LAA@FORSUB
FORSUB   PROC    FAR
         PUBLIC  FORSUB
         PUSH    BP
         MOV     BP,SP
; NO OTHER ROUTINE CALLED SO
; DON'T BOTHER SETTING UP TRACKBACK

         MOV     AX,LAA@FORSUB
         MOV     DS,AX

; SS:BP HAS ADDRESS OF PARAMETER
; BLOCK (FRAME).

; LOAD ADDRESS OF THE FIRST VALUE INTO DS:SI
         LDS     SI,[BP].PARMVAL1

; LOAD THE FIRST VALUE
         MOV     AX,[SI]

; LOAD ADDRESS OF THE SECOND VALUE INTO DS:SI
         LDS     SI,[BP].PARMVAL2

; ADD SECOND VALUE TO FIRST IN AX
         ADD     AX,[SI]

; LOAD ADDRESS OF THE THIRD VALUE INTO DS:SI
         LDS     SI,[BP].PARMVAL3
```

```
; ADD THIRD VALUE TO RUNNING SUM IN AX
        ADD     AX,[SI]

; LOAD ADDRESS OF THE VARIABLE WHERE THE SUM
; WILL BE PLACED INTO THE DS:SI
        LDS     SI,[BP].PARSUM

; STORE THE SUM IN THE ACTUAL ARGUMENT
; ON THE FORTRAN CALL STATEMENT
        MOV     [SI],AX

; DIVIDE THE SUM BY 3 AND LEAVE RESULT IN AL
        MOV     CL,3
        IDIV    CL

; CLEAR THE REMAINDER
        XOR     AH,AH

; LOAD ADDRESS OF THE VARIABLE WHERE THE MEAN
; WILL BE PLACED INTO DS:SI
        LDS     SI,[BP].PARMEAN

; STORE THE MEAN IN THE ACTUAL ARGUMENT
; ON THE FORTRAN CALL STATEMENT
        MOV     [SI],AX

; RETURN TO FORTRAN
; (DON'T NEED TO COPY BP TO
; SP SINCE SP STILL EQUALS BP)
        POP     BP
        RET     20      ; POP PARAMETERS

FORSUB  ENDP
P@FORSUB ENDS
STACK   SEGMENT  WORD  'STACK'  STACK
STACK   ENDS
        END
```

In the 8088, the stack grows from higher addresses to lower addresses. That is, as data is placed on the stack (pushed), the stack pointer register decreases. As data is removed (popped) from the stack, the stack pointer register increases. The following stack diagram illustrates this:

Stack Diagram



You should note the following points when coding your assembler language subprogram:

- Assembler language subprograms are always considered FAR procedures. That is, IBM FORTRAN/2 uses a Long call instruction to transfer control to your subprogram. This means that you must declare your assembler language subprogram as a FAR procedure by using the PROC statement in Macro Assembler/2. See "Sample Assembler Language Subprogram" on page 8-10 for an example.
- Assembler language subprograms are always considered EXTERN procedures. That is, when you use the name of a function or subroutine in a IBM FORTRAN/2 program, that name is passed to the linker as an EXTRN FAR.
- When you link your FORTRAN and assembler language modules together, the linker tries to find a PUBLIC symbol that matches the EXTRN name from the IBM FORTRAN/2 program. This means that you must declare the name of your assembler language subprograms as a PUBLIC name using the PUBLIC directive in Macro Assembler/2. See "Sample Assembler Language Subprogram" on page 8-10 for an example of this.

- IBM FORTRAN/2 uses "call by reference", plus 2-byte lengths for character arguments, by pushing the 4-byte addresses of an actual argument onto the stack. This occurs even when the actual argument is an expression or constant. When the actual argument is a variable or subscripted variable or array, the address of this item is pushed onto the stack, followed by pushing the length for character arguments. If the actual argument is an expression or constant, the actual argument is evaluated and the result is placed in a temporary location. The address of this location is pushed onto the stack, followed by pushing the length for a character argument. If the actual argument is a procedural parameter (subroutine or function parameter), the entry point of the procedure is pushed onto the stack.
- The $VAL function can be used to pass INTEGER*2 and INTEGER*4 values using "call by value".
- You should link your assembler language subprogram after at least one object module has been compiled with IBM FORTRAN/2. That is, an object module produced from the IBM FORTRAN/2 compiler should appear first in the object module files list supplied to the linker.
- If you plan to include segments that do not normally exist when using IBM FORTRAN/2, you must be careful that these segments are linked in a way that will not interfere with the normal FORTRAN segment.

Hints for Debugging an Assembler Language Program

1. You can include the instruction INT 3 as the first executable instruction of your assembly language subprogram. You can then run the linked .EXE file under the DOS DEBUG program. Your program will break point at this instruction and you can use the DOS program to analyze the logic of your subprogram. Use the G= or T= options of DEBUG to step over the INT 3 to continue execution of your assembler language subprogram. An .EXE file with this INT 3 instruction can only be run under the DOS DEBUG program.
2. Always check the link map to make sure that your segments are ordered correctly. All the CODE segments should be grouped together.
3. Your stack segment has to be the last segment of your run module. Check your link map to verify this.
4. Make sure you leave the stack, BP, SP, and SS registers in the state they were in when your subprogram received control.
5. Make sure your RET instruction removes the arguments.

Compiler Generated and Library Segment Names

The following is a list of the segment names. This includes the class and group names (where applicable) as well as the segment name. The combine and align types for the segments are also given.

All code segments (segments with CLASS 'CODE') must have access rights of EXECUTEREAD. All data segments (segments that do not have CLASS 'CODE') must have access rights of READWRITE and NONSHARED except where noted. None of the segments require or should have I/O privilege level (IOPL). Other attributes may be chosen as desired.

The program entry point is at the address defined by the PUBLIC symbol given by the name (uppercase) in the PROGRAM statement. If there is no PROGRAM statement, F@MAIN (uppercase) is used for this PUBLIC symbol.

The entry point to an external procedure (function or subroutine) is at the address defined by the PUBLIC symbol with the same name (uppercase) as given in the FUNCTION, SUBROUTINE, or ENTRY statement.

For a common block in a BLOCK DATA subprogram, a PUBLIC symbol having the same name (uppercase) as the common block is defined at the beginning of the first segment of the common block. For blank common, the name _BLANK (uppercase) is used for the name of this PUBLIC symbol.

Compiler Generated Segments for a Program Unit

In the following,

> Name is the name (uppercase) given in the PROGRAM, SUBROUTINE or FUNCTION statement of the program unit. For a main program unit without a PROGRAM statement, F@MAIN (uppercase) is used for the name.
>
> Comm is the name (uppercase) of a common block. For a blank common, _BLANK (uppercase) is used for comm.

Only Common Data segments are generated for a BLOCK DATA subprogram.

Segment	Name	Class	Group	Combine	Align
Code	P@name	CODE	G@name[1]	PUBLIC	BYTE
Constants[2]	C@name	CODE	G@name	PUBLIC	PARA
Local Data 1					
First segment	LAA@name	F@DATA	n/a	PUBLIC	PARA
Other segments	LAB@name[3]	F@DATA	n/a	PUBLIC	PARA
Local Data 2 (character temporaries)[4]					
First segment	TAA@name	F@DATA	n/a	PUBLIC	BYTE
Other segments	TAB@name[3]	F@DATA	n/a	PUBLIC	BYTE
Common Data					
First segment	comm	F@DATA	n/a	PUBLIC	PARA
Other segments	comm@AA[3]	F@DATA	n/a	PUBLIC	PARA
Debug Tables (/T option only)					
First segment	DAA@name[3]	DEBUG	n/a	PUBLIC	PARA
Other segments	DAB@name[3]	DEBUG	n/a	PUBLIC	PARA
null 1	F@@DATA	DATA	DGROUP	PUBLIC	BYTE
null 2	??SEG	n/a	n/a	PUBLIC	BYTE
Stack[5]	STACK	STACK	n/a	STACK	WORD

Notes:

1. The group name is only generated if a Constants segment (C@name) is also generated. See note 2.
2. The Constants segment is only generated when the size of the constant area generated by the compiler plus the size of the local data area is greater than 65536. The constants in this segment are considered to be part of the Code segment (because of the group G@name).
3. Each subsequent name carries the next character of the alphabet in the sequence AA, AB, AC, AD, ..., AZ, BA, BB, ..., BZ, CA,..., ZZ.
4. These segments are only generated if character functions are called or if character expressions are passed as arguments.
5. For subprograms, the size of the stack segment is equal to the stack space needed by the subprogram. For a main program, the size of the stack segment is equal to the stack space needed by the main program plus 1KB. The additional 1KB provides stack

space for calling intrinsic functions, Debug, emulator, operating system, and other language routines (such as assembly language). The size of the stack space is given in the compilation listing of the program unit (use the /S option to obtain a program unit summary). See "Stack" on page 8-22 for more information on stack space allocation.

Library—Intrinsic Segments

Segment	Name	Class	Group	Combine	Align
Code	F@ICODE	CODE	n/a	PUBLIC	BYTE/WORD
Data 1[1]	F@IDATA1	F@DATA	n/a	COMMON	WORD
Data 2[1]	F@IDATA2	F@DATA	n/a	COMMON	WORD
Data 3[1]	F@IDATA3	F@DATA	n/a	COMMON	WORD
Data 4[1]	F@IDATA4	F@DATA	n/a	COMMON	WORD
Data 5[1]	F@IDATA5	F@DATA	n/a	COMMON	WORD
Stack[2]	STACK	STACK	n/a	STACK	WORD
null	??SEG	n/a	n/a	PUBLIC	PARA

Notes:

1. These segments are only included as needed.
2. This stack segment has zero length. The stack provided by the main program is used. See "Stack" on page 8-22 for more information on stack space allocation.

Library—Runtime Segments

Segment	Name	Class	Group	Combine	Align
Code 1	F@RT	CODE	n/a	PUBLIC	BYTE/WORD
Code 2	F@MALC	CODE	n/a	PUBLIC	BYTE
Code 3[1]	F@RCODE3	CODE	n/a	PUBLIC	WORD
Code 4[1]	F@RCODE4	CODE	n/a	PUBLIC	BYTE
Code 5[2]	F@DOS	CODE	n/a	PUBLIC	BYTE
Data 1[1]	F@OSDAT	F@DATA	n/a	PUBLIC	WORD
Data 2	F@IOCOM	F@DATA	n/a	COMMON	WORD
Data 3	F@IOSYSI	F@DATA	n/a	PUBLIC	BYTE
Data 4	F@MALD	F@DATA	n/a	PUBLIC	WORD
Data 5[1]	F@RDATA5	F@DATA	n/a	COMMON	WORD
Stack[3]	STACK	STACK	n/a	STACK	WORD
null	??SEG	n/a	n/a	PUBLIC	PARA

Notes:

1. This segment is only included as needed.
2. This segment is only included if linked for DOS mode.
3. This stack segment has zero length. An internal stack of the required size is used.

Library—Emulator Segments (/N option only)

Segment	Name	Class	Group	Combine	Align
Code	F@ECODE	CODE	n/a	PUBLIC	WORD
Data	F@EDATA	F@DATA	n/a	PUBLIC	WORD
Stack*	STACK	STACK	n/a	STACK	WORD
null	??SEG	n/a	n/a	PUBLIC	PARA

***Note:** This stack segment has zero length. The stack provided by the main program is used. See "Stack" on page 8-22 for more information on stack space allocations.

Library—Debug Segments (/T option only)

Segment	Name	Class	Group	Combine	Align
Code	F@DCODE	CODE	n/a	PUBLIC	WORD
Data 1	F@DDATA1	F@DATA	n/a	COMMON	WORD
Data 2	F@DDATA2[1]	F@DATA	n/a	PUBLIC	WORD
Stack[2]	STACK	STACK	n/a	STACK	WORD
null	??SEG	n/a	n/a	PUBLIC	PARA

Notes:

1. READONLY and SHARED attributes are allowed and preferred.
2. This stack segment has zero length. The stack provided by the main program is used. See "Stack" on page 8-22 for more information on stack space allocation.

Stack

The linker determines the default size of the stack by adding up the stack space requirements of all the program units. This default is normally adequate.

It may be desirable to override this default using the /STACK option of the LINK command or the STACKSIZE statement in a module-definition file for a link. For more information on the /STACK option see "/STACK" on page 4-40.

The size of the stack segment can be reduced when the default creates a stack segment that is larger than is needed. The actual stack requirements can be calculated by taking the maximum of the sum of program unit stack requirements for each call path through the program. This maximum should be increased by enough bytes to call intrinsic functions, Debug, emulator, operating system, and other language routines (the main program supplies an additional 1KB for this purpose). The size of the stack space requirements for a

program unit is given in the compilation listing of the program unit (use the /S option to obtain a program unit summary). For example, if the call structure of a program is:

```
           Main Program (4E + 1K)
                       |
         +-----------+-----------+
         |           |           |
      SubA (46)   SubB (52)   FunC (3A)
         |           |           |
      SubD (1C)   FuncE (38)  SubF (2A)
```

The sums for each call path are B0 + 1KB, D8 + 1KB, and B2 + 1KB. Therefore, the maximum is D8 + 1KB. The default would be 23C + 1KB. The extra 1KB can be reduced if intrinsics, Debug, emulator, operating system calls, or other language routines are not used. The extra 1KB may need to be increased if other language routines are called which require additional stack space.

The stack segment may need to be increased when dynamic link modules are used which require additional stack space. The stack space required by the dynamic link routines is not included in determining the default stack space allocation.

The stack segment should be reduced to the minimum (2 bytes) when creating dynamic link modules. Stack allocated for the dynamic link module will not be used (the stack supplied by the calling program is used instead).

Creating a stack that is too large wastes memory and may prevent a program from being loaded if it is too large. If this occurs, the size of the stack should be reduced.

A stack that is too small may cause the program to overrun the allocated stack space. In DOS mode, the program may give unpredictable results. In protect mode, a stack overrun message will be given by the operating system.

OS and Application Programming Interface Procedures

IBM FORTRAN/2 allows you to directly access OS external procedures. These procedures may be Application Programming Interface procedures, or procedures you supply. You may provide procedures to be accessed this way, provided they follow the calling and return conventions listed in "Calling OS Procedures" on page 8-26. Accessing Application Programming Interface procedures is supported only when linking under OS/2. If linking under DOS, you must supply all OS external procedures.

Application Programming Interface Functions

All Application Programming Interface function calls are supported.

The following Application Programming Interface functions have limited use (though all can be called from IBM FORTRAN/2):

DosExitList Routine must be defined in some language other than IBM FORTRAN/2.

DosSetSigHandler Routine must be defined in some language other than IBM FORTRAN/2.

DosSetVec Routine must be defined in some language other than IBM FORTRAN/2.

DosCreateThread Routine is recommended to be written in some other language other than IBM FORTRAN/2.

Note also that some Application Programming Interface functions are supposed to be used only by the session manager. These functions should therefore only be used if the session manager is written in IBM FORTRAN/2 (which is not recommended).

Linking to Application Programming Interface functions is accomplished by using the DOSCALLS.LIB import library. The BIND utility and the API.LIB library are used to convert such an EXE file to an EXE file that can execute on both OS/2 and DOS (in this case, Application Programming Interface functions should be restricted to Family Application Programming Interface functions).

When linking to Application Programming Interface functions, you must specify the DOSCALLS.LIB library.

Note: When making a portable application, proper use of the **/N** and **/Y** compiler options should also be considered.

Examples

The following example shows how to use DOSWRITE.

```
      OS EXTERNAL DOSWRITE
      INTEGER*2 DOSWRITE, BYTESWRITTEN, RC, FILEHANDLE
      INTEGER*2 BUFFERLENGTH
      CHARACTER*100 BUFFERAREA
.
.
.
C     Get a value for FILEHANDLE from DOSOPEN
      BUFFERLENGTH = 6
      BUFFERAREA = 'RECORD'
      RC = DOSWRITE($val(FILEHANDLE), BUFFERAREA,
     1    $VAL(BUFFERLENGTH), BYTEWRITTEN)
```

The following example shows how to increase the number of files that can be opened from the default of 20 to a value of 30.

```
      OS EXTERNAL DOSSETMAXFH
      INTEGER*2 DOSSETMAXFH,NUMBERHANDLES,RC

.
.
.
      NUMBERHANDLES = 30
      RC = DOSSETMAXFH($VAL(NUMBERHANDLES))
```

Support routines are supplied to help set up the arguments to or use the returned values of an Application Programming Interface function. See the Language Reference Manual for information about the following:

Copy Memory Routines (COPYMEM) These can be used to access data obtained from an Application Programming Interface procedure, or to create data to be passed to an Application Programming Interface procedure.

Inter-language Call Interface Routine (FORTILC) This can be modified to supply a desired interface to a routine that was dynamically loaded at runtime.

API functions can be used to load dynamic link libraries at runtime and obtain the addresses of routines in the modules. The address would be returned in an INTEGER*4 variable.

If the calling conventions are the same for IBM FORTRAN/2 or for OS Externals, an interface routine can be written in IBM FORTRAN/2 to call the routine indirectly. For example, to call a subroutine:

```
      INTEGER*4 rtn
.
.
.
C     rtn = address of dynamically loaded at runtime routine.
      CALL IRTN($Val(rtn),arg1, arg2,...)
.
.
.
      END

      SUBROUTINE IRTN(drtn, dmy1, dmy2,...)
      EXTERNAL drtn
      CALL drtn(dmy1, dmy2,...)
      END
```

If the routine is an OS EXTERNAL, prefix EXTERNAL with OS in the above example.

If the calling conventions are not the same as for IBM FORTRAN/2 or for OS Externals, the FORTILC routines (contained in FORTILC.ASM on the "LINK_RUN"master 5¼" diskette or the "INSTALL" master 3½" diskette) can be modified to convert the callers (IBM FORTRAN/2 or OS) parameter conventions, as needed, to agree with the conventions used by the dynamic link routine. The address of the routine (in an Integer*4 variable) can then be passed by value using the $VAL intrinsic function to invoke the routine.

FORTILC is discussed in Appendix E, "Additional Routines" of *IBM FORTRAN/2 Language Reference*.

Calling OS Procedures

If this procedure returns a value in the AX register, the procedure should be called as a function reference; it must also be typed INTEGER*2 explicitly or implicitly. In all other cases, the procedure should be called using a CALL statement.

OS procedures have the same calling argument and return conventions as for IBM FORTRAN/2 procedures, except that lengths of character arguments are not pushed and only INTEGER*2 function results are allowed.

If character arguments are to be passed to an OS procedure, the procedure must be specified in an OS EXTERNAL statement in the calling program.

Note: The OS EXTERNAL statement is an extension to the ANSI X3.9-1978 FORTRAN 77 Standard.

Arguments are passed on the stack. They are pushed on the stack in left to right order, as given in the procedure reference arguments. Arguments are passed by address in selector: offset format, or by value. Only integer expressions may be passed by value. Passing an expression by value is specified by using the special $VAL intrinsic function.

Using $VAL, INTEGER*2 expressions are passed as word values on the stack. Using $VAL, INTEGER*4 expressions are passed as double word values on the stack.

The ASCIIZ intrinsic function can be used to create character values that end with a null (0 value) character.

A far call is generated to reference the subroutine or function.

The referenced procedure removes the arguments from the stack.

A procedure referenced as a function should return an INTEGER*2 result in the AX register.

Arguments must match expected parameters in order, type, address/value, and number.

The IBM Math Co-Processor stack will be empty on call and should be empty on return. The state of IBM Math Co-Processor numeric processor should be the same on exit as it was on call.

The state of the processor should be the same on exit as it was on call except for the AX register, which may be altered by the referenced procedure. In particular, make sure the direction flag is clear.

Index

Special Characters

A

B

C

D

E

F

G

M

N

O

P

Q

T

U

V

W

IBM United Kingdom
International Products Limited
PO Box 41, North Harbour
Portsmouth, PO6 3AU
England

Printed in Denmark by Interprint A/S

IBM